



PYTHON 3

Proste wprowadzenie
do fascynującego
świata programowania

Z E D A . S H A W

Helion 

Tytuł oryginału: Learn Python 3 the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code (Zed Shaw's Hard Way Series)

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-4142-5

Authorized translation from the English language edition, entitled: LEARN PYTHON 3 THE HARD WAY: A VERY SIMPLE INTRODUCTION TO THE TERRIFYINGLY BEAUTIFUL WORLD OF COMPUTERS AND CODE; ISBN 0134692888; by Zed A. Shaw; published by Pearson Education, Inc. Copyright © 2017 by Zed A. Shaw.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. Polish language edition published by HELION S.A. Copyright © 2018.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pyt3pw_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to!» Nasza społeczność](#)

Spis treści

Przedmowa	15
Ulepszenia w edycji Python 3	15
Trudna droga jest łatwiejsza	16
Czytanie i pisanie	16
Dbłość o szczegóły	16
Dostrzeganie różnic	16
Pytaj, pytaj i pytaj	17
Nie przeklejaj	17
Uwagi na temat ćwiczeń i wytrwałości	17
Podziękowania	18
Ćwiczenie 0. Konfiguracja	20
macOS	20
macOS. Co powinienś zobaczyć	21
Windows	21
Windows. Co powinienś zobaczyć	22
Linux	23
Linux. Co powinienś zobaczyć	23
Szukanie informacji w internecie	24
Ostrzeżenia dla początkujących	24
Alternatywne edytory tekstów	25
Ćwiczenie 1. Dobry pierwszy program	28
Co powinienś zobaczyć	29
Zrób to sam	31
Typowe pytania	31
Ćwiczenie 2. Komentarze i znaki kratki	34
Co powinienś zobaczyć	34
Zrób to sam	34
Typowe pytania	35
Ćwiczenie 3. Liczby i działania algebraiczne	36
Co powinienś zobaczyć	37
Zrób to sam	37
Typowe pytania	37

Ćwiczenie 4. Zmienne i nazwy	40
Co powinienś zobaczyć	41
Zrób to sam	41
Typowe pytania	42
Ćwiczenie 5. Więcej zmiennych i drukowania	44
Co powinienś zobaczyć	44
Zrób to sam	45
Typowe pytania	45
Ćwiczenie 6. Łącuchy znaków i tekst	46
Co powinienś zobaczyć	47
Zrób to sam	47
Popsuj kod	47
Typowe pytania	48
Ćwiczenie 7. Więcej drukowania	50
Co powinienś zobaczyć	50
Zrób to sam	50
Popsuj kod	51
Typowe pytania	51
Ćwiczenie 8. Drukowanie, drukowanie	52
Co powinienś zobaczyć	52
Zrób to sam	53
Typowe pytania	53
Ćwiczenie 9. Drukowanie, drukowanie, drukowanie	54
Co powinienś zobaczyć	54
Zrób to sam	55
Typowe pytania	55
Ćwiczenie 10. Co to było?	56
Co powinienś zobaczyć	57
Sekwencje ucieczki	57
Zrób to sam	58
Typowe pytania	58
Ćwiczenie 11. Zadawanie pytań	60
Co powinienś zobaczyć	60
Zrób to sam	61
Typowe pytania	61

Ćwiczenie 12. Wyświetlanie odpowiedzi dla użytkownika	62
Co powinienś zobaczyć	62
Zrób to sam	62
Typowe pytania	63
Ćwiczenie 13. Parametry, rozpakowywanie i zmienne	64
Chwileczkę! Funkcjonalności mają jeszcze inną nazwę	65
Co powinienś zobaczyć	65
Zrób to sam	66
Typowe pytania	66
Ćwiczenie 14. Znak zachęty i przekazywanie zmiennej	68
Co powinienś zobaczyć	69
Zrób to sam	69
Typowe pytania	69
Ćwiczenie 15. Czytanie z plików	72
Co powinienś zobaczyć	73
Zrób to sam	73
Typowe pytania	74
Ćwiczenie 16. Czytanie i zapisywanie plików	76
Co powinienś zobaczyć	77
Zrób to sam	78
Typowe pytania	78
Ćwiczenie 17. Więcej plików	80
Co powinienś zobaczyć	80
Zrób to sam	81
Typowe pytania	82
Ćwiczenie 18. Nazwy, zmienne, kod i funkcje	84
Co powinienś zobaczyć	85
Zrób to sam	86
Typowe pytania	86
Ćwiczenie 19. Funkcje i zmienne	88
Co powinienś zobaczyć	89
Zrób to sam	89
Typowe pytania	89

Ćwiczenie 20. Funkcje i pliki	92
Co powinienś zobaczyć	93
Zrób to sam	93
Typowe pytania	93
Ćwiczenie 21. Funkcje mogą coś zwracać	96
Co powinienś zobaczyć	97
Zrób to sam	97
Typowe pytania	98
Ćwiczenie 22. Czego nauczyłeś się do tej pory?	100
Miej świadomość, czego się uczysz	100
Ćwiczenie 23. Łącuchy znaków, bajty i kodowanie znaków	102
Wstępne badanie kodu	102
Przełączniki, konwencje i rodzaje kodowania	104
Analizujemy dane wyjściowe	106
Analizujemy kod	106
Bawimy się kodowaniem	109
Popsuj kod	109
Ćwiczenie 24. Więcej praktyki	110
Co powinienś zobaczyć	111
Zrób to sam	111
Typowe pytania	111
Ćwiczenie 25. Jeszcze więcej praktyki	112
Co powinienś zobaczyć	113
Zrób to sam	114
Typowe pytania	115
Ćwiczenie 26. Gratulacje, rozwiąż test!	116
Typowe pytania	117
Ćwiczenie 27. Zapamiętywanie logiki	118
Wyrażenie logiczne	119
Tablice prawdy	119
Typowe pytania	120
Ćwiczenie 28. Ćwiczymy logikę boolowską	122
Co powinienś zobaczyć	124
Zrób to sam	124
Typowe pytania	124

Ćwiczenie 29. Co, jeśli...	126
Co powinienś zobaczyć	126
Zrób to sam	127
Typowe pytania	127
Ćwiczenie 30. Else oraz if	128
Co powinienś zobaczyć	129
Zrób to sam	129
Typowe pytania	129
Ćwiczenie 31. Podejmowanie decyzji	130
Co powinienś zobaczyć	131
Zrób to sam	131
Typowe pytania	131
Ćwiczenie 32. Pętle i listy	134
Co powinienś zobaczyć	135
Zrób to sam	136
Typowe pytania	136
Ćwiczenie 33. Pętle while	138
Co powinienś zobaczyć	139
Zrób to sam	139
Typowe pytania	140
Ćwiczenie 34. Uzyskiwanie dostępu do elementów list	142
Zrób to sam	143
Ćwiczenie 35. Gałęzie i funkcje	144
Co powinienś zobaczyć	145
Zrób to sam	146
Typowe pytania	146
Ćwiczenie 36. Projektowanie i debugowanie	148
Zasady dotyczące instrukcji if	148
Zasady dotyczące pętli	148
Wskazówki dotyczące debugowania	149
Praca domowa	149
Ćwiczenie 37. Przegląd symboli	150
Słowa kluczowe	150
Typy danych	151

Sekwencje ucieczki	152
Formatowanie łańcuchów znaków w starym stylu	152
Operatory	153
Czytanie kodu	154
Zrób to sam	155
Typowe pytania	155
Ćwiczenie 38. Operacje na listach	156
Co powinienś zobaczyć	157
Co mogą robić listy	158
Kiedy używać list	159
Zrób to sam	159
Typowe pytania	160
Ćwiczenie 39. Słowniki, piękne słowniki	162
Przykład słownika	163
Co powinienś zobaczyć	164
Co mogą robić słowniki	165
Zrób to sam	166
Typowe pytania	166
Ćwiczenie 40. Moduły, klasy i obiekty	168
Moduły są jak słowniki	168
Klasy są jak moduły	169
Obiekty są jak import	170
Różne sposoby pobierania elementów	171
Pierwszy przykład klasy	172
Co powinienś zobaczyć	172
Zrób to sam	172
Typowe pytania	173
Ćwiczenie 41. Uczymy się mówić obiektowo	174
Ćwiczmy słówka	174
Ćwiczmy zdania	174
Ćwiczenie łączone	175
Test z czytania	175
Tłumaczenie ze zdań na kod	177
Poczytaj jeszcze więcej kodu	178
Typowe pytania	178

Ćwiczenie 42. Jest, ma, obiekty i klasy	180
Jak to wygląda w kodzie	181
Na temat class Nazwa(object)	183
Zrób to sam	183
Typowe pytania	184
Ćwiczenie 43. Podstawowa analiza obiektowa i projekt	186
Analiza prostego silnika gry	187
Opisz lub rozrysuj problem	187
Wydrebnij kluczowe pojęcia i zbadaj je	188
Utwórz hierarchię klas i mapę obiektów dla koncepcji	188
Zakoduj klasy i napisz test, aby je uruchomić	189
Powtórz i udoskonal	191
Z góry do dołu i z dołu do góry	191
Kod gry Goci z planety Percal 25	192
Co powinieneś zobaczyć	198
Zrób to sam	198
Typowe pytania	199
Ćwiczenie 44. Porównanie dziedziczenia i kompozycji	200
Co to jest dziedziczenie	200
Dziedziczenie domyślne	201
Bezpośrednie nadpisanie	202
Zmiana zachowania przed lub po	202
Połączenie wszystkich trzech sposobów	203
Dlaczego super()	205
Używanie super() z __init__	205
Kompozycja	205
Kiedy używać dziedziczenia, a kiedy kompozycji	207
Zrób to sam	207
Typowe pytania	207
Ćwiczenie 45. Tworzysz grę	210
Ocenianie napisanej gry	210
Styl funkcji	211
Styl klas	211
Styl kodu	212
Dobre komentarze	212
Oceń swoją grę	213

Ćwiczenie 46. Szkielet projektu	214
Konfiguracja w systemach macOS i Linux	214
Konfiguracja w systemie Windows 10	215
Tworzenie szkieletu katalogu projektów	217
Ostateczna struktura katalogów	218
Testowanie konfiguracji	219
Używanie szkieletu	220
Wymagany quiz	220
Typowe pytania	220
Ćwiczenie 47. Zautomatyzowane testowanie	222
Pisanie przypadku testowego	222
Wytyczne testowania	224
Co powinienś zobaczyć	224
Zrób to sam	225
Typowe pytania	225
Ćwiczenie 48. Zaawansowane wprowadzanie danych przez użytkownika	226
Nasz leksykon gry	226
Rozkładanie zdań na części	227
Krotki leksykonu	227
Skanowanie danych wejściowych	227
Wyjątki i liczby	227
Wyzwanie „najpierw przygotuj testy”	228
Co powinienś przetestować	229
Zrób to sam	231
Typowe pytania	231
Ćwiczenie 49. Tworzenie zdań	232
Dopasowywanie i podglądanie	232
Gramatyka zdania	233
Słowo o wyjątkach	233
Kod parsera	233
Zabawa z parserem	236
Co powinienś przetestować	237
Zrób to sam	237
Typowe pytania	237

Ćwiczenie 50. Twoja pierwsza strona internetowa	238
Instalowanie frameworku flask	238
Tworzenie prostego projektu „Witaj, świecie”	238
Co się tutaj dzieje	240
Naprawianie błędów	240
Tworzenie podstawowych szablonów	241
Zrób to sam	243
Typowe pytania	243
Ćwiczenie 51. Pobieranie danych wejściowych z przeglądarki	246
Jak działa sieć	246
Jak działają formularze	248
Tworzenie formularzy HTML	249
Tworzenie szablonu układu	251
Pisanie zautomatyzowanych testów dla formularzy	252
Zrób to sam	254
Popsuj kod	254
Ćwiczenie 52. Początek Twojej gry internetowej	256
Refaktoryzacja gry z ćwiczenia 43.	256
Tworzenie silnika	261
Twój egzamin końcowy	263
Typowe pytania	264
Następne kroki	266
Jak uczyć się dowolnego języka programowania	267
Porada starego programisty	270
Dodatek. Przyspieszony kurs wiersza poleceń	272
Wprowadzenie: zamknij się i zacznij używać powłoki	272
Jak korzystać z tego dodatku	273
Będziesz musiał zapamiętywać rzeczy	273
Konfiguracja	274
Zadanie	274
Czego się nauczyłeś	275
Zadanie dodatkowe	275
Ścieżki, foldery, katalogi (pwd)	277
Zadanie	277
Czego się nauczyłeś	278
Zadanie dodatkowe	278

Jeśli się zgubisz	279
Zadanie	279
Czego się nauczyłeś	279
Tworzenie katalogu (mkdir)	279
Zadanie	279
Czego się nauczyłeś	280
Zadanie dodatkowe	281
Zmienianie katalogu (cd)	281
Zadanie	281
Czego się nauczyłeś	284
Zadanie dodatkowe	284
Listowanie katalogu (ls)	285
Zadanie	285
Czego się nauczyłeś	287
Zadanie dodatkowe	288
Usuwanie katalogu (rmdir)	288
Zadanie	288
Czego się nauczyłeś	290
Zadanie dodatkowe	290
Poruszanie się po katalogach (pushd, popd)	290
Zadanie	291
Czego się nauczyłeś	292
Zadanie dodatkowe	293
Tworzenie pustych plików (touch/New-Item)	293
Zadanie	293
Czego się nauczyłeś	294
Zadanie dodatkowe	294
Kopiowanie pliku (cp)	294
Zadanie	294
Czego się nauczyłeś	296
Zadanie dodatkowe	297
Przenoszenie pliku (mv)	297
Zadanie	297
Czego się nauczyłeś	298
Zadanie dodatkowe	298
Przeglądanie pliku (less/more)	299
Zadanie	299
Czego się nauczyłeś	300
Zadanie dodatkowe	300

Strumieniowanie pliku (cat)	300
Zadanie	300
Czego się nauczyłeś	301
Zadanie dodatkowe	301
Usuwanie pliku (rm)	301
Zadanie	301
Czego się nauczyłeś	303
Zadanie dodatkowe	303
Wyjście z terminala (exit)	303
Zadanie	303
Czego się nauczyłeś	303
Zadanie dodatkowe	303
Następne kroki z wierszem poleceń	304
Zasoby dla uniksowej powłoki bash	304
Skorowidz	305

Przedmowa

Ta prosta książka ma sprawić, żebyś zaczął programować. Wykorzystano tu tak zwaną technikę *instruktażu*. Instruktaż polega na tym, że instruuje Cię, jak masz wykonać sekwencję kontrolowanych ćwiczeń, które mają na celu budowanie umiejętności przez powtarzanie. Technika ta bardzo dobrze sprawdza się w przypadku początkujących, którzy nie mają żadnej wiedzy i muszą zdobyć podstawowe umiejętności, zanim będą w stanie zrozumieć bardziej złożone kwestie. Jest używana we wszystkich dziedzinach od sztuk walki przez muzykę aż po podstawową matematykę i umiejętności czytania.

Książka ta ma stanowić instruktaż posługiwania się językiem Python; ma pomóc w powolnym budowaniu i ugruntowywaniu umiejętności przy wykorzystaniu technik, takich jak ćwiczenie i zapamiętywanie, a następnie wyjaśnianie, jak stosować te umiejętności do coraz trudniejszych problemów. Gdy skończysz czytać tę książkę, będziesz znać narzędzia potrzebne do nauki bardziej złożonych tematów dotyczących programowania. Lubię mówić, że moja książka daje „czarny pas w programowaniu”. Oznacza to, że poznasz podstawy na tyle dobrze, aby rozpocząć naukę programowania.

Jeśli ciężko popracujesz, nie będziesz się spieszyć i posiadasz te umiejętności, nauczysz się kodować.

Ulepszenia w edycji Python 3

Książka *Python 3* korzysta teraz z Pythona 3.6. Znormalizowałem treść dla tej wersji Pythona, ponieważ zawiera ona nowy, ulepszony system formatowania łańcuchów znaków, który jest łatwiejszy w użyciu niż ten z poprzedniej 4. wersji (lub 3., zapomniałem, było ich tak wiele). Początkujący mają z reguły kilka problemów z Pythonem 3.6, ale w tej książce wyjaśnię Ci, jak poruszać się po tych kwestiach. Szczególnie dotkliwym problemem jest to, że Python 3.6 ma bardzo słaby system komunikatów o błędach w niektórych kluczowych obszarach; postaram się te problemy wyjaśnić.

W tej książce oprócz wykonywania ćwiczeń wprowadziłem również zadania związane z psuciem, a następnie naprawianiem kodu z każdego ćwiczenia. Ta umiejętność nazywa się „debugowaniem”. Uczy, jak rozwiązywać napotykanne problemy, ale także uświadamia, w jaki sposób Python uruchamia programy, które tworzysz. Celem tej nowej metodologii jest zbudowanie mentalnego modelu sposobu, w jaki Python uruchamia Twój kod, dzięki czemu łatwiej będzie Ci zorientować się, dlaczego jest popsuty. Nauczysz się również wielu użytecznych sztuczek związanych z debugowaniem uszkodzonego oprogramowania.

Ostatnią kwestią jest to, że edycja Python 3 w pełni obsługuje system Microsoft Windows 10 od początku do końca. Poprzednia edycja książki skupiała się głównie na systemach uniksowych, takich jak macOS i Linux, a Windows był uwzględniany raczej jako spóźniona refleksja. Gdy zacząłem pisać o edycji Python 3, Microsoft zaczął poważnie traktować narzędzia *open source* i programistów, zatem trudno było zignorować ich systemy jako poważną platformę programistyczną Pythona.

Trudna droga jest łatwiejsza

Z pomocą tej książki będziesz robił niewiarygodnie proste rzeczy, które wykonują wszyscy programiści, aby nauczyć się języka programowania. Zasady są proste.

1. Przeczytaj każde ćwiczenie.
2. Wpisz *dokładnie* każdy fragment kodu.
3. Spraw, by zadziałał.

To wszystko! Na początku będzie to bardzo trudne, ale trzymaj się tych zasad. Jeśli podczas czytania tej książki wykonasz każde ćwiczenie, poświęcając na to godzinę lub dwie każdego wieczora, będziesz miał dobre podstawy, aby przejść do kolejnej książki o Pythonie i kontynuować naukę. Książka nie zamieni Cię w programistę z dnia na dzień, ale na pewno znajdziesz się na właściwej ścieżce do nauki kodowania.

Pisząc tę książkę, postanowiłem nauczyć Cię trzech najważniejszych umiejętności, które musi posiadać początkujący programista; są to czytanie i pisanie, dbałość o szczegóły i dostrzeganie różnic.

Czytanie i pisanie

Jeżeli masz kłopoty z wpisywaniem, będziesz miał problem z nauką kodowania szczególnie wtedy, jeśli trudność sprawia Ci wpisywanie dość dziwnych znaków w kodzie źródłowym. Bez tej prostej umiejętności nie będziesz w stanie nauczyć się nawet najbardziej podstawowych rzeczy dotyczących działania oprogramowania.

Wpisywanie przykładów kodu i uruchamianie ich pomoże Ci nauczyć się nazw symboli, poznać ich wpisywanie i zmusi do czytania tego języka.

Dbałość o szczegóły

Jedną z umiejętności odróżniających dobrych programistów od złych jest dbałość o szczegóły. To naprawdę wyróżnia dobrego fachowca w każdym zawodzie. Musisz zwracać uwagę na najdrobniejsze szczegóły Twojej pracy, ponieważ w przeciwnym razie umkną Ci ważne elementy tego, co tworzysz. W przypadku programowania prowadzi to do powstawania pełnego błędów oprogramowania i trudnych do użycia systemów.

Pracując nad tą książką i *dokładnie* powielając każdy przykład, będziesz ćwiczyć swój umysł w skupianiu się na bieżąco na szczegółach tego, co robisz.

Dostrzeganie różnic

Bardzo ważną umiejętnością (którą większość programistów rozwija z biegiem czasu) jest zdolność wizualnego dostrzegania różnic między pewnymi rzeczami. Doświadczony programista może wziąć dwa niewiele różniące się od siebie fragmenty kodu i natychmiast wskazać różnice. Programiści wymyślili nawet narzędzia, które jeszcze bardziej to ułatwiają, ale nie będziemy używać żadnego z nich. Najpierw musisz mozolnie wyćwiczyć swój umysł, a dopiero potem możesz zacząć korzystać z narzędzi.

Gdy będziesz wykonywał ćwiczenia, wpisując każde z nich, będziesz popełniał błędy. To nieuniknione. Nawet zaprawieni w boju programiści zrobiliby kilka błędów. Twoim zadaniem jest porównywać to, co napisałeś, z tym, co jest wymagane, i usuwać wszystkie różnice. W ten sposób nauczysz się zauważać pomyłki, błędy i inne problemy.

Pytaj, pytaj i pytaj

Podczas pisania kodu będziesz generować błędy. „Błąd” (ang. *bug*, czyli robak) oznacza wadę, usterkę lub problem z napisanym przez Ciebie kodem. Legenda głosi, że to określenie wzięło się od prawdziwej ćmy, która wleciała do wnętrza jednego z pierwszych komputerów, powodując jego nieprawidłowe działanie. Naprawienie tego wymagało „odrobaczenia” (ang. *debugging*) komputera. W świecie oprogramowania istnieje *dużo* błędów. Bardzo dużo.

Jak ta pierwsza ćma, Twoje błędy będą ukryte gdzieś w kodzie i musisz je znaleźć. Nie możesz po prostu usiąść przed ekranem komputera, wpatrywać się w słowa, które napisałeś, i mieć nadzieję, że odpowiedź sama się objawi. W ten sposób nie uzyskasz żadnych dodatkowych informacji, a ich potrzebujesz. Musisz wstać i poszukać ćmy.

W tym celu musisz „przesłuchać” swój kod i zapytać go, co się dzieje, lub spojrzeć na problem z innej perspektywy. W tej książce często będę powtarzał, żebyś „przestał się gapić i zapytał”. Pokażę Ci, jak zmusić kod, żeby powiedział wszystko, co „wie”, co się dzieje i co zmienić, by uzyskać rozwiązanie. Pokażę Ci również, jak patrzeć na kod z różnych perspektyw, aby uzyskać więcej informacji i zrozumieć problem.

Nie przeklejaj

Musisz *wpisać* każde z tych ćwiczeń ręcznie. Jeśli masz zamiar przeklejać kod, równie dobrze możesz w ogóle nie wykonywać ćwiczeń. Celem tych ćwiczeń jest trenowanie rąk, mózgu i umysłu w czytaniu, pisaniu i postrzeganiu kod. Jeśli będziesz kopiował i wklejał, wyeliminujesz efektywność tych lekcji.

Uwagi na temat ćwiczeń i wytrwałości

Gdy Ty studiujesz programowanie, ja uczę się grać na gitarze. Ćwiczę codziennie przez co najmniej dwie godziny. Przez godzinę gram skale, akordy i arpeggia, a potem uczę się teorii muzyki, ćwiczę słuch, piosenki i wszystko, co tylko mogę. Są dni, kiedy uczę się gry na gitarze i studium muzykę przez osiem godzin, bo mam na to ochotę i jest to fajne. Dla mnie powtarzanie ćwiczeń jest czymś naturalnym i właśnie w ten sposób uczę się różnych rzeczy. Wiem, że jeśli chcę być w czymś dobry, muszę ćwiczyć codziennie nawet wtedy, jeśli tego dnia jestem „do bani” (co się często zdarza) lub jest to trudne. Próbuje dalej, a w końcu stanie się to łatwiejsze i przyjemniejsze.

W czasie, który upłynął pomiędzy napisaniem przeze mnie książek *Learn Python the Hard Way* i *Learn Ruby the Hard Way*, odkryłem rysowanie i malowanie. Zakochałem się w sztukach wizualnych w wieku 39 lat i poświęcałem każdy dzień na ich studiowanie w taki sam sposób, w jaki uczyłem się gry na gitarze, muzyki i programowania. Zgromadziłem książki z materiałami instruktażowymi, zrobiłem to, co napisano w tych książkach, malowałem codziennie i skupiałem się na czerpaniu radości z procesu uczenia się. Nie jestem „artystą”, nie jestem

w tym nawet dość dobry, ale teraz mogę powiedzieć, że potrafię rysować i malować. Ta sama metoda, której uczę Cię w tej książce, miała zastosowanie do moich przygód ze sztuką. Jeśli rozłożysz problem na niewielkie ćwiczenia i lekcje, i będziesz wykonywał je codziennie, możesz nauczyć się wykonywać prawie wszystko. Gdy skupisz się na powolnych postępach i czerpaniu radości z procesu uczenia się, odniesiesz korzyści bez względu na to, jak dobry w tym będziesz.

Podczas studiowania tej książki i kontynuowania nauki programowania pamiętaj, że wszystko, co ma jakąś wartość, jest z początku trudne. Może jesteś osobą, która boi się porażki, więc rezygnujesz przy pierwszych pojawiających się trudnościach. Może nigdy nie nauczyłeś się samodyscypliny, więc nie potrafisz robić niczego, co jest „nudne”. Być może powiedziano Ci, że jesteś „utalentowany”, więc nigdy nie próbujesz niczego, przez co mógłbyś wyjść na głupka i zniszczyć mit cudownego dziecka. A może jesteś przesadnie ambitny i niesprawiedliwie porównujesz się do kogoś takiego jak ja, kto programuje od ponad 20 lat.

Bez względu na to, z jakiego powodu chciałbyś zrezygnować, *nie poddawaj się*. Zmusz się do pracy. Jeśli napotkasz podrozdział „Zrób to sam”, którego zadań nie potrafisz wykonać, lub lekcję, której po prostu nie rozumiesz, pomiń je i wróć do nich później. Po prostu przejdź dalej, ponieważ w przypadku programowania dzieje się pewna bardzo dziwna rzecz. Na początku niczego nie rozumiesz. Wszystko wydaje się dziwne, podobnie jak przy nauce jakiegokolwiek ludzkiego języka. Walczysz ze słowami, nie wiesz, co znaczą poszczególne symbole, i wszystko jest bardzo dezorientujące. A potem pewnego dnia *TRACH!* — Twój mózg zaskakuje i nagle „załapujesz”. Jeśli będziesz kontynuował ćwiczenia i próbował je zrozumieć, załapiesz to. Możesz nie będziesz mistrzem kodowania, ale przynajmniej zrozumiesz, jak działa programowanie.

Jeśli się poddasz, nigdy nie osiągniesz tego punktu. Zderzysz się z pierwszą dezorientującą rzeczą (a na początku wszystko jest dezorientujące) i zatrzymasz się. Jeśli będziesz próbował, dalej wpisywał kod, starał się zrozumieć i czytać o tym, w końcu załapiesz, o co chodzi. Jeżeli przestudiujesz całą tę książkę i nadal nie będziesz wiedział, jak kodować, przynajmniej powiesz, że dałeś sobie szansę. Możesz powiedzieć, że starałeś się z całych sił i jeszcze trochę bardziej, ale nie wyszło. Jednak przynajmniej próbowałeś. Możesz być z tego dumny.

Podziękowania

Chciałbym podziękować Angeli za pomoc przy pierwszych dwóch wersjach tej książki. Bez niej prawdopodobnie w ogóle nie zadałbym sobie trudu, żeby ją dokończyć. Zająła się zadiustowaniem pierwszego szkicu i niezmiernie mnie wspierała, gdy pisałem.

Chciałbym także podziękować Gregowi Newmanowi za wykonanie oryginalnej okładki, Brianowi Shumate'owi za początkowe projekty strony internetowej i wszystkim osobom, które czytają tę książkę i poświęciły swój czas, żeby przesłać mi opinie i poprawki.

Dziękuję Wam.

Konfiguracja

To ćwiczenie nie zawiera kodu. Należy je ukończyć po to, żeby zainstalować na komputerze obsługę Pythona. Powinieneś wykonać wszystkie instrukcje możliwie najdokładniej.

OSTRZEŻENIE! Jeśli nie wiesz, jak używać programu PowerShell w systemie Windows, terminala w systemie macOS lub powłoki bash w Linuksie, musisz najpierw nauczyć się nimi posługiwać. Przed przejściem do właściwych ćwiczeń powinieneś najpierw wykonać ćwiczenia z dodatku „Przyspieszony kurs wiersza poleceń”.

macOS

Aby ukończyć to ćwiczenie, wykonaj następujące kroki.

1. Otwórz w przeglądarce stronę <https://www.python.org/downloads/release/python-360/> i pobierz wersję programu zatytułowaną *Mac OS X 64-bit/32-bit installer*. Zainstaluj ją, tak jak każdy inny program.
2. Przejdź do strony <https://atom.io>, pobierz edytor tekstu Atom i zainstaluj. Jeśli ten edytor tekstu Ci nie odpowiada, na końcu tego ćwiczenia znajdziesz podrozdział „Alternatywne edytory tekstu”.
3. Zapewnij sobie łatwy dostęp do edytora Atom, umieszczając skrót do programu w Docku.
4. Znajdź swój program terminala.
5. Terminal również umieść w Docku.
6. Uruchom program terminala. Nie będzie on wyglądać oszałamiająco.
7. W terminalu uruchom `python3.6`. Aby uruchomić dowolny program w terminalu, należy wpisać jego nazwę i nacisnąć przycisk *Return*.
8. Wpisz `quit()` i naciśnij *Return*, aby wyjść z Pythona.
9. Powinieneś wrócić do tego samego nagłówka wiersza poleceń, który był wyświetlony przed wpisaniem polecenia `python`. Jeśli tak się nie stanie, spróbuj dowiedzieć się, dlaczego tak się stało.
10. Naucz się tworzyć katalogi w terminalu. Utwórz katalog i nazwij go na przykład *lpthw*.
11. Naucz się zmieniać katalogi w terminalu.

12. Użyj edytora tekstu, aby utworzyć plik *test.txt* w przygotowanym wcześniej katalogu. W tym celu wybierz z menu opcję *File/New File* (w polskiej wersji *Plik/Nowy plik*), a następnie wybierz *Save* lub *Save As...* (*Zapisz lub Zapisz jako...*) i wskaż odpowiednią lokalizację do zapisania pliku.
13. Wróć do terminala, używając klawiatury do przełączania okien.
14. W terminalu wyświetl listę katalogów za pomocą polecenia `ls`, aby zobaczyć swój nowo utworzony plik.

macOS. Co powinieneś zobaczyć

Wykonałem te czynności w terminalu na moim komputerze macOS. Twój listing może zawierać trochę inne informacje niż mój, ale nie musisz się tym przejmować — zasadniczo powinien wyglądać podobnie.

```
$ python3.6
Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
~ $ mkdir lpthw
~ $ cd lpthw
lpthw $ ls
# ... W tym miejscu użyj edytora tekstu do edycji pliku test.txt...
lpthw $ ls
test.txt
lpthw $
```

Windows

1. Otwórz w przeglądarce stronę <https://atom.io>, pobierz edytor tekstu Atom i zainstaluj. Aby to zrobić, nie musisz posiadać uprawnień administratora systemu.
2. Zapewnij sobie łatwy dostęp do edytora Atom, umieszczając skrót do programu na pulpicie i (lub) na pasku zadań. Obie opcje są dostępne podczas instalacji. Jeśli nie możesz uruchomić edytora Atom, ponieważ Twój komputer nie jest wystarczająco szybki, na końcu tego ćwiczenia znajdziesz podrozdział „Alternatywne edytory tekstu”.
3. Uruchom program PowerShell z menu *Start*. W tym celu najpierw wyszukaj program, a następnie naciśnij przycisk *Enter*, aby go uruchomić.
4. Dla wygody umieść skrót do tego programu na pulpicie i (lub) na pasku zadań.
5. Po uruchomieniu program PowerShell (będę go później nazywał terminalem) nie będzie wyglądać oszałamiająco.
6. Pobierz Pythona 3.6 ze strony <https://www.python.org/downloads/release/python-360/> i zainstaluj. Pamiętaj, aby podczas instalacji zaznaczyć pole **Add Python 3.6 to PATH** (dodaj Pythona 3.6 do zmiennej środowiskowej PATH).

7. Uruchom polecenie python w programie PowerShell (terminalu). Aby uruchomić dowolny program w terminalu, wystarczy po prostu wpisać jego nazwę i nacisnąć *Enter*. Jeśli po wpisaniu polecenia python program nie zostanie uruchomiony, musisz zainstalować go ponownie i upewnić się, że zaznaczyłeś pole *Add Python 3.6 to PATH*. Jest to bardzo małe pole, więc poszukaj go uważnie.
8. Wpisz `quit()` i naciśnij *Enter*, aby wyjść z Pythona.
9. Powinieneś wrócić do tego samego nagłówka wiersza poleceń, który był wyświetlony przed wpisaniem polecenia python. Jeśli tak się nie stanie, spróbuj dowiedzieć się, dlaczego tak się stało.
10. Naucz się tworzyć katalogi w programie PowerShell (terminalu). Utwórz katalog i nazwij go na przykład *lpthw*.
11. Naucz się zmieniać katalogi w programie PowerShell (terminalu).
12. Użyj edytora tekstu, aby utworzyć plik *test.txt* w przygotowanym wcześniej katalogu. W tym celu wybierz z menu opcję *Plik/Nowy*, a następnie wybierz *Zapisz* lub *Zapisz jako...* i wskaż odpowiednią lokalizację do zapisania pliku.
13. Wróć do programu PowerShell (terminala), używając klawiatury do przełączania okien. Jeśli nie wiesz, jak to zrobić, poszukaj informacji w internecie.
14. W programie PowerShell (terminalu) wyświetl listę katalogów, aby zobaczyć swój nowo utworzony plik.

Od tej chwili, gdy będę używał określeń „terminal” lub „powłoka”, będę miał na myśli aplikację *PowerShell* i właśnie jej powinieneś używać. Kiedy będę instruował Cię, żebyś uruchomił python3.6, możesz po prostu wpisać polecenie python.

Windows. Co powinieneś zobaczyć

```
> python
>>> quit()
> mkdir lpthw
> cd lpthw
... W tym miejscu użyj edytora tekstu, aby utworzyć plik test.txt katalogu lpthw ...
>
> dir
Volume in drive C is
Volume Serial Number is 085C-7E02

Directory of C:\Documents and Settings\nazwa_użytkownika\lpthw

04.05.2010  23:32    <DIR>          .
04.05.2010  23:32    <DIR>          ..
04.05.2010  23:32                6 test.txt
                1 File(s)                6 bytes
                2 Dir(s)  14 804 623 360 bytes free

>
```

Twój listing może zawierać trochę inne informacje niż mój, ale nie musisz się tym przejmować — zasadniczo powinien wyglądać podobnie.

Linux

Linux jest zróżnicowanym systemem operacyjnym z wieloma różnymi sposobami instalacji oprogramowania. Zakładam, że jeśli używasz systemu Linux, wiesz, jak instalować pakiety. Oto instrukcje.

1. Użyj menedżera pakietów Linuksa do zainstalowania Pythona 3.6, a jeśli nie możesz tego zrobić, pobierz pliki źródłowe ze strony <https://www.python.org/downloads/release/python-360/> i skompiluj ze źródła.
2. Użyj menedżera pakietów Linuksa do zainstalowania edytora tekstu Atom. Jeśli ten edytor tekstu Ci nie odpowiada, na końcu tego ćwiczenia znajdziesz podrozdział „Alternatywne edytory tekstu”.
3. Zapewnij sobie łatwy dostęp do edytora Atom, umieszczając skrót do programu w menu menedżera okien (lub panelu uruchamiania).
4. Znajdź swój program terminala. Może to być GNOME Terminal, Konsole lub xterm.
5. Terminal również dodaj do panelu uruchamiania.
6. Uruchom program terminala. Nie będzie wyglądać oszałamiająco.
7. W terminalu uruchom `python3.6`. Aby uruchomić dowolny program w terminalu, należy wpisać jego nazwę i nacisnąć przycisk *Enter*. Jeśli nie możesz uruchomić polecenia `python3.6`, spróbuj wpisać po prostu `python`.
8. Wpisz `quit()` i naciśnij *Enter*, aby wyjść z Pythona.
9. Powinieneś wrócić do tego samego nagłówka wiersza poleceń, który był wyświetlony przed wpisaniem polecenia `python`. Jeśli tak się nie stanie, spróbuj dowiedzieć się, dlaczego tak się stało.
10. Naucz się tworzyć katalogi w terminalu. Utwórz katalog i nazwij go na przykład `lpthw`.
11. Naucz się zmieniać katalogi w terminalu.
12. Użyj edytora tekstu, aby utworzyć plik `test.txt` w przygotowanym wcześniej katalogu. W tym celu wybierz z menu opcję *File/New File* (w polskiej wersji *Plik/Nowy plik*), a następnie wybierz *Save* lub *Save As...* (*Zapisz lub Zapisz jako...*) i wskaż odpowiednią lokalizację do zapisania pliku.
13. Wróć do terminala, używając klawiatury do przełączania okien. Jeśli nie wiesz, jak to zrobić, poszukaj informacji w internecie.
14. W terminalu wyświetl listę katalogów, aby zobaczyć swój nowo utworzony plik.

Linux. Co powinieneś zobaczyć

```
$ python
>>> quit()
$ mkdir lpthw
$ cd lpthw
```

```
# ... W tym miejscu użyj edytora tekstu do edycji pliku test.txt ...  
$ ls  
test.txt  
$
```

Twój listing może zawierać trochę inne informacje niż mój, ale nie musisz się tym przejmować — zasadniczo powinien wyglądać podobnie.

Szukanie informacji w internecie

Podczas lektury tej książki w dużej mierze będziesz uczył się wyszukiwać w internecie tematy związane z programowaniem. Jeśli napiszę, żebyś „poszukał tego w internecie”, Twoim zadaniem będzie użycie wyszukiwarki do znalezienia odpowiedzi. Zamiast od razu podawać odpowiedź, będę Cię zmuszał do poszukiwań, ponieważ chcę, żebyś był niezależnym uczniem, który nie będzie potrzebował już mojej książki, kiedy ją przeczyta. Jeśli potrafisz znaleźć w internecie odpowiedzi na Twoje pytania, jesteś o krok bliżej do tego, żebym przestał być Ci potrzebny, a taki jest mój cel.

Dzięki wyszukiwarkom, takim jak Google, możesz łatwo znaleźć wszystko, co każę Ci znaleźć. Jeśli powiem: „poszukaj w internecie funkcji list pythona”, wykonaj po prostu następujące czynności.

1. Otwórz stronę <http://google.com>.
2. Wpisz: **funkcje list python3**.
3. Poczytaj wyświetlone strony internetowe w celu znalezienia najlepszej odpowiedzi.

Ostrzeżenia dla początkujących

Na tym kończymy to ćwiczenie. Może ono okazać się trudne, co zależy od Twojej znajomości komputera. Jeśli tak właśnie jest, poświęć trochę czasu na uważną lekturę i uporanie się z tym ćwiczeniem, ponieważ dopóki nie będziesz w stanie wykonać tych podstawowych czynności, samo programowanie również będzie sprawiało Ci trudności.

Jeśli ktoś powie Ci, żebyś poprzestał na konkretnym ćwiczeniu z tej książki lub pominął niektóre ćwiczenia, powinieneś zignorować taką osobę. Każdy, kto chce ukryć przed Tobą wiedzę lub — co gorsza — narzucić Ci, żebyś tę wiedzę przyjmował jedynie od niego, a nie zdobywał własnym wysiłkiem, stara się uzależnić poziom Twoich umiejętności od siebie. Nie słuchaj takich osób i nadal wykonuj ćwiczenia, aby nauczyć się samodzielnie pozyskiwać wiedzę.

W końcu dlaczego jakiś programista każe Ci używać systemu macOS lub Linux? Jeśli lubi czcionki i typografię, powie Ci, żebyś korzystał z komputera macOS. Jeśli lubi kontrolę i ma ogromną brodę (choć niekoniecznie musi być to programista płci męskiej), poleci Ci zainstalować Linux. W takim przypadku również nie kieruj się sugestiami i używaj dowolnego działającego komputera. Potrzebujesz jedynie edytora tekstu, terminala i Pythona.

Ta konfiguracja ma na celu pomóc Ci w wykonywaniu w niezawodny sposób trzech rzeczy podczas pracy nad ćwiczeniami. Oto one.

1. *Pisanie* ćwiczenia przy użyciu edytora tekstu.
2. *Uruchamianie* ćwiczenia, które napisałeś.
3. *Naprawianie* go, gdy coś się popsuje.
4. Powtarzanie powyższych czynności.

Wszystko poza tym wprowadzi tylko niepotrzebne zamieszanie, więc trzymaj się planu.

Alternatywne edytory tekstów

Edytory tekstów są bardzo ważne dla programisty, ale jako początkujący potrzebujesz jedynie prostego **programistycznego edytora tekstu**. Takie edytory różnią się oprogramowaniem służącym do pisania opowieści i książek, ponieważ uwzględniają unikatowe potrzeby kodu komputerowego. W tej książce polecam edytor tekstu Atom, ponieważ jest bezpłatny i działa na niemal każdej platformie. Atom może jednak nie działać zbyt dobrze na Twoim komputerze, więc oto kilka innych możliwości do wypróbowania.

Nazwa edytora	Platforma	Link do pobrania
Visual Studio Code	Windows, macOS, Linux	https://code.visualstudio.com
Notepad++	Windows	https://notepad-plus-plus.org
gEdit	Linux, macOS, Windows	https://github.com/GNOME/gedit
Textmate	macOS	https://github.com/textmate/textmate
SciTE	Windows, Linux	http://www.scintilla.org/SciTE.html
jEdit	Linux, macOS, Windows	http://www.jedit.org

Edytory w tabeli zostały wymienione w takiej kolejności, w jakiej najprawdopodobniej zadziałają. Pamiętaj, że te projekty mogą zostać porzucone, umrzeć w sposób naturalny lub przestać działać na Twoim komputerze. Jeśli spróbujesz jednego i nie będzie działać, spróbuj następnego. Systemy operacyjne w wierszach kolumny *Platforma* również są wymienione w kolejności określającej największe prawdopodobieństwo zadziałania programu, więc jeśli korzystasz z systemu Windows, powinieneś szukać edytorów, dla których Windows jest wymieniony jako pierwszy w danym wierszu tej kolumny.

Jeśli wiesz już, jak używać edytorów Vim lub Emacs, możesz z nich skorzystać. Jeżeli nigdy nie używałeś tych edytorów, raczej ich unikaj. Programiści mogą przekonywać do używania edytorów Vim lub Emacs, ale to jedynie wytrąci Cię z obranego toru. Twoim celem jest nauka Pythona, a nie wymienionych edytorów. Jeśli spróbujesz użyć Vima i nie będziesz wiedział, jak wyjść z programu, wpisz :q! lub ZZ. Jeśli ktoś polecił Ci korzystać z edytora Vim i nie przekazał nawet tej informacji, wiesz już, dlaczego nie powinieneś go słuchać.

Podczas wykonywania ćwiczeń z tej książki nie używaj zintegrowanego środowiska programistycznego (ang. *Integrated Development Environment* — IDE). Poleganie na IDE oznacza, że nie będziesz mógł pracować z nowymi językami programowania, dopóki jakaś firma nie

zdecyduje się sprzedać Ci IDE dla tego języka. Nie będziesz mógł więc używać tego nowego języka, dopóki nie stanie się on na tyle popularny, żeby dostarczyć bazy lukratywnych klientów. Jeśli nauczysz się pracować jedynie z programistycznym edytorem tekstu (takim jak Vim, Emacs, Atom i podobnymi), nie będziesz uzależniony od zewnętrznych dostawców oprogramowania. Środowiska IDE są przyjemne w niektórych sytuacjach (takich jak praca z olbrzymią bazą istniejącego kodu), ale uzależnienie od nich ograniczy Twój rozwój.

Nie powinieneś również używać IDLE. Ma ono poważne ograniczenia w sposobie działania i nie jest zbyt dobrym elementem oprogramowania. Potrzebujesz jedynie prostego edytora tekstu, powłoki i Pythona.

Dobry pierwszy program

OSTRZEŻENIE! Jeśli pominąłeś ćwiczenie 0, nie korzystasz z tej książki we właściwy sposób. Próbujesz używać IDLE lub IDE? W ćwiczeniu 0 zazaczyłem, żebyś nie używał tych środowisk, więc nie powinieneś tego robić. Zatem proszę, wróć do tego ćwiczenia i przeczytaj je.

Powinieneś poświęcić sporo czasu na ćwiczenie 0, żeby nauczyć się, jak zainstalować edytor tekstu, uruchomić go, uruchomić terminal i pracować z oboma narzędziami. Jeśli tego nie zrobiłeś, nie kontynuuj bieżącego ćwiczenia, ponieważ nie spędzisz miło czasu. To jedyny raz, kiedy rozpoczynam ćwiczenie ostrzeżeniem, że nie powinieneś niczego pomijać lub wybiegać do przodu.

Wpisz poniższy tekst w pojedynczym pliku o nazwie `ex1.py`. Python działa najlepiej z plikami z rozszerzeniem `.py`.

ex1.py

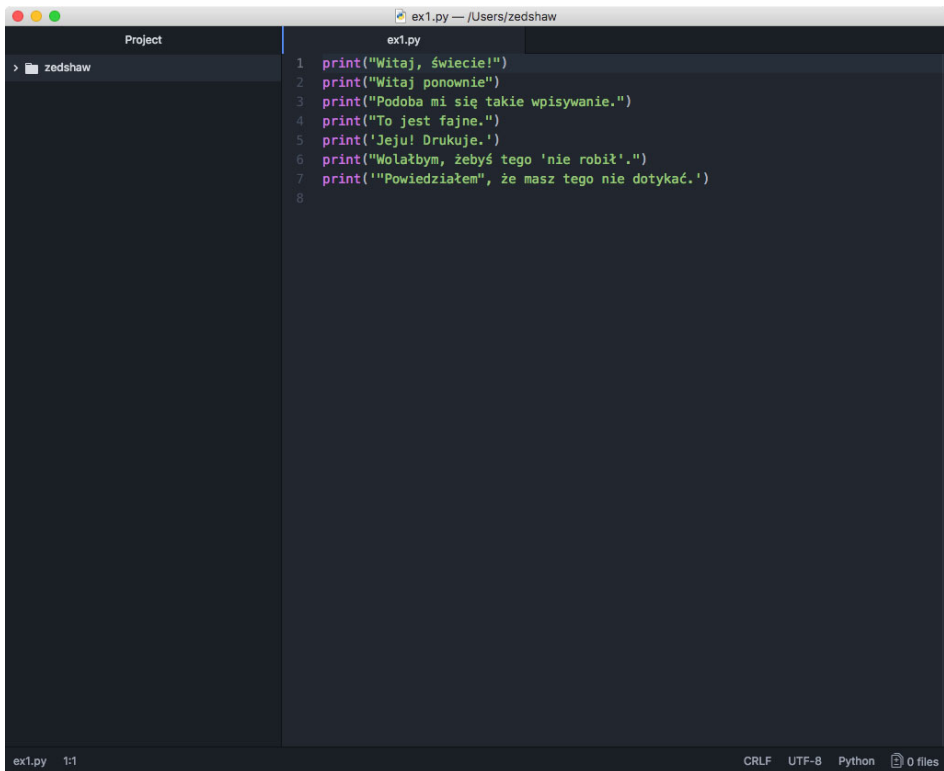
```
1 print("Witaj, świecie!")
2 print("Witaj ponownie")
3 print("Podoba mi się takie wpisywanie.")
4 print("To jest fajne.")
5 print('Jeju! Drukuje.')
6 print("Wolałbym, żebyś tego 'nie robił'.")
7 print("Powiedziałem", że masz tego nie dotykać.")
```

Okno edytora tekstu Atom powinno wyglądać podobnie na wszystkich platformach (zobacz rysunek na następnej stronie).

Nie przejmuj się, jeśli Twój edytor nie wygląda dokładnie tak samo — powinien wyglądać dość podobnie. Możesz mieć trochę inny nagłówek okna, może nieco inne kolory, a w lewym panelu nie będzie wyświetlać się nazwa projektu `zedshaw`, ale zamiast tego wyświetli się katalog, którego użyłeś do zapisywania plików. Wszystkie te różnice są w porządku.

Podczas tworzenia tego pliku powinieneś pamiętać o następujących kwestiach.

1. Nie wpisywałem numerów linii po lewej stronie. Stosuję je w listingach, żebym mógł odwoływać się do konkretnych linii, na przykład: „Zobacz linię 5.”. W skryptach Pythona nie wpisuje się numerów linii.
2. Na początku każdej linii znajduje się polecenie `print`, które wygląda dokładnie tak samo jak w moim pliku `ex1.py`. „Dokładnie” oznacza dokładnie, a nie mniej więcej tak samo. Aby kod zadziałał, musi się zgadzać każdy pojedynczy znak. Kolor nie ma znaczenia, tylko wpisywane znaki.



```
ex1.py — /Users/zedshaw
Project
> zedshaw
ex1.py
1 print("Witaj, świecie!")
2 print("Witaj ponownie")
3 print("Podoba mi się takie wpisywanie.")
4 print("To jest fajne.")
5 print('Jeju! Drukuje.')
6 print("Wolałbym, żebyś tego 'nie robił'.")
7 print('Powiedziałem, że masz tego nie dotykać.')
8
ex1.py 1:1 CRLF UTF-8 Python 0 files
```

W terminalu w systemie macOS lub Linux uruchom ten plik, wpisując następujące polecenie:

```
python3.6 ex1.py
```

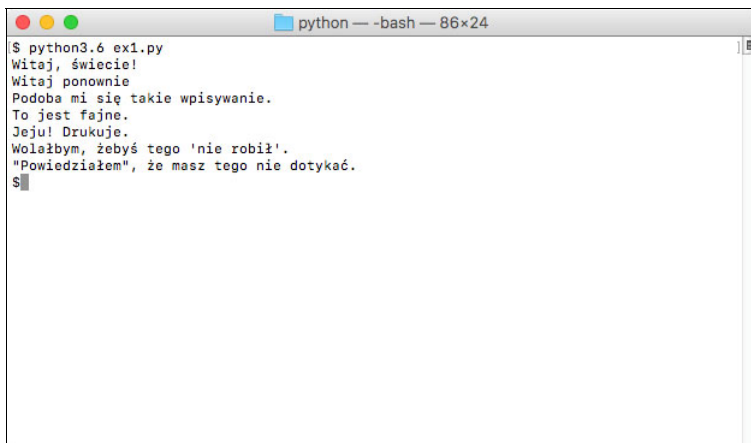
Pamiętaj, że w systemie Windows zawsze wpisujesz python zamiast python3.6, tak jak poniżej:

```
python ex1.py
```

Jeśli zrobiłeś wszystko prawidłowo, powinieneś zobaczyć to samo, co przedstawiono w podrozdziale „Co powinieneś zobaczyć” tego ćwiczenia. Jeśli tego nie widzisz, zrobiłeś coś źle. Nie, komputer się nie pomylił.

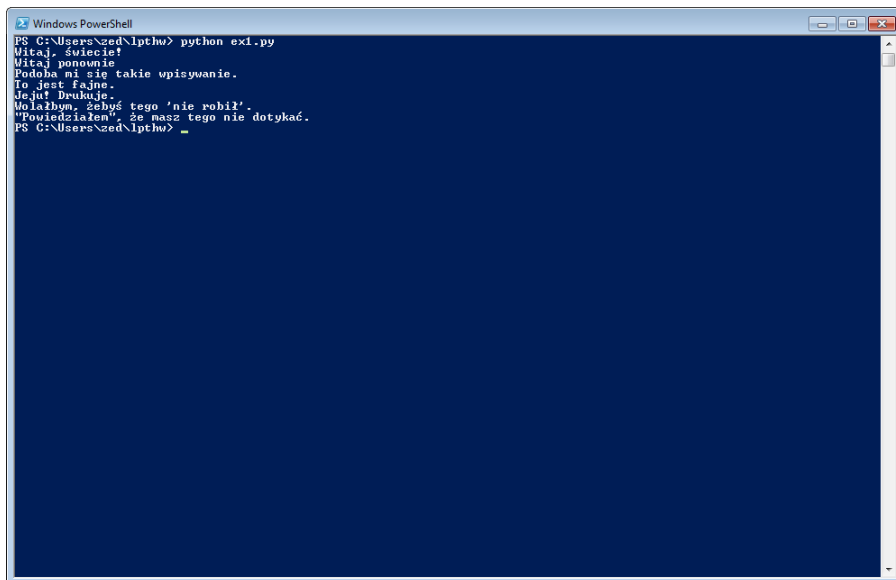
Co powinieneś zobaczyć

Oto, co powinieneś zobaczyć w terminalu w systemie macOS.

A terminal window titled 'python — bash — 86x24' showing the execution of a Python script. The output is as follows:

```
$ python3.6 ex1.py
Witaj, świecie!
Witaj ponownie
Podoba mi się takie wpisywanie.
To jest fajne.
Jeju! Drukuje.
Wolałbym, żebyś tego 'nie robił'.
"Powiedziałem", że masz tego nie dotykać.
$
```

To powinieneś zobaczyć w PowerShell w systemie Windows.

A Windows PowerShell window titled 'Windows PowerShell' showing the execution of a Python script. The output is as follows:

```
PS C:\Users\zed\lpthw> python ex1.py
Witaj, świecie!
Witaj ponownie
Podoba mi się takie wpisywanie.
To jest fajne.
Jeju! Drukuje.
Wolałbym, żebyś tego 'nie robił'.
"Powiedziałem", że masz tego nie dotykać.
PS C:\Users\zed\lpthw> _
```

Przed poleceniem `python3.6 ex1.py` mogą być wyświetlane różne nazwy, ale najważniejsze jest to, żebyś po wpisaniu tego polecenia zobaczył dane wyjściowe takie same jak moje.

Jeśli wystąpi błąd, będzie to wyglądać następująco.

```
$ python3.6 python/ex1.py
File "python/ex1.py", line 3
    print("Podoba mi się takie wpisywanie.
      ^
SyntaxError: EOL while scanning string literal
```

Ważne jest, abyś potrafił odczytywać komunikaty o błędach, ponieważ będziesz popełniał wiele takich błędów. Nawet ja popełniam ich sporo. Przyjrzyjmy się temu linia po linii.

1. Wpisaliśmy nasze polecenie w terminalu, aby uruchomić skrypt `ex1.py`.
2. Python poinformował nas, że plik `ex1.py` ma błąd w linii 3.
3. Wyświetlił tę linię kodu, abyśmy mogli ją zobaczyć.
4. Następnie umieścił znak wstawiania (^), aby wskazać, gdzie jest problem. Zwróciłeś uwagę na brakujący znak podwójnego cudzysłowu (")?
5. Na koniec wyświetlił komunikat `SyntaxError` i odpowiedział, co może być błędem. Zazwyczaj te komunikaty są bardzo tajemnicze, ale jeśli skopiujesz dany tekst do wyszukiwarki, znajdziesz kogoś innego, kto napotkał ten sam błąd, i prawdopodobnie dowiesz się, jak go naprawić.

Zrób to sam

Podrozdziały zatytułowane „Zrób to sam” zawierają zadania, które powinienes *spróbować* wykonać. Jeśli na razie nie jesteś w stanie poradzić sobie z danym ćwiczeniem, pomiń je i wróć do niego później.

W tym ćwiczeniu spróbuj zrobić trzy rzeczy.

1. Spraw, aby Twój skrypt wydrukował jeszcze jedną linię.
2. Spraw, aby Twój skrypt wydrukował tylko jedną z linii.
3. Umieść znak kratki (#) na początku linii. Co on powoduje? Spróbuj się dowiedzieć, do czego służy ten znak.

Odtąd nie będę już wyjaśniał, na czym polega każde ćwiczenie, chyba że dane ćwiczenie będzie się różnić od pozostałych.

OSTRZEŻENIE! Znak kratki jest nazywany również „krzyżykiem”, „hashem”, „płotkiem” i tak dalej. Wybierz tę nazwę, która Cię relaksuje.

Typowe pytania

Są to *faktyczne* pytania, które prawdziwi studenci zadali podczas wykonywania tego ćwiczenia.

Czy mogę używać IDLE? Nie. Powinienes używać terminala w systemach macOS i Linux oraz programu PowerShell w systemie Windows, tak jak robię w tym ćwiczeniu. Jeśli nie wiesz, jak korzystać z tych narzędzi, możesz przeczytać dodatek „Przyspieszony kurs wiersza poleceń”.

Jak uzyskać kolory w edytorze? Najpierw zapisz plik jako plik z rozszerzeniem `.py`, na przykład `ex1.py`. Wtedy podczas wpisywania pojawiają się kolory.

Gdy próbuję uruchomić plik *ex1.py*, otrzymuję komunikat `SyntaxError: invalid syntax`. Prawdopodobnie uruchomiłeś Pythona, a następnie próbujesz ponownie wpisać polecenie `python` w celu uruchomienia pliku. Zamknij terminal, uruchom go ponownie i od razu wpisz jedynie `python3.6 ex1.py`.

Otrzymuję komunikat `can't open file 'ex1.py': [Errno 2] No such file or directory`. Musisz znajdować się w tym samym katalogu, co utworzony plik. Aby przejść do właściwego katalogu, użyj najpierw polecenia `cd`. Jeśli zapisałeś plik na przykład w lokalizacji *lpthw/ex1.py*, przejdź do niej za pomocą polecenia `cd lpthw/` i dopiero wtedy wpisz polecenie `python3.6 ex1.py`. Jeśli nie wiesz, co to wszystko oznacza, zapoznaj się z dodatkiem „Przyspieszony kurs wiersza poleceń”.

Mój plik nie działa. Po prostu otrzymuję z powrotem nagłówki wiersza poleceń bez żadnych danych wyjściowych. Najprawdopodobniej potraktowałeś zbyt dosłownie kod z mojego pliku *ex1.py* i uznałeś, że `print("Witaj, świecie!")` oznacza, że w pliku należy wpisać tylko `"Witaj, świecie!"` bez słowa `print`. Twój plik musi być *dokładnie* taki sam jak mój.

Komentarze i znaki kratki

Komentarze w Twoich programach są bardzo ważne. Informują, co robi dany fragment kodu, i są używane do wyłączania części programu, jeśli trzeba je tymczasowo usunąć.

Komentarzy w Pythonie używa się w poniższy sposób.

ex2.py

```
1  # Komentarz; ułatwia późniejsze czytanie programu.
2  # Wszystko, co znajduje się po znaku #, jest przez Pythona ignorowane.
3
4  print("Mógłbym napisać taki kod.") # a komentarz po kodzie będzie ignorowany
5
6  # Komentarza możesz również użyć do "wyłączenia", czyli wykomentowania kodu:
7  # print("To nie zostanie uruchomione.")
8
9  print("To zostanie uruchomione.")
```

Od tej pory będę pisać kod właśnie w taki sposób. Powinieneś zrozumieć, że nie wszystko musi być dosłowne. Twój ekran i program mogą wyglądać inaczej, ale najważniejszy jest umieszczony w pliku tekst, który piszesz w edytorze. W rzeczywistości mógłbym pracować z dowolnym edytorem tekstu i rezultat byłby taki sam.

Co powinieneś zobaczyć

Ćwiczenie 2. — sesja

```
$ python3.6 ex2.py
Mógłbym napisać taki kod.
To zostanie uruchomione.
```

Tym razem również nie pokażę Ci zrzutów ekranu wszystkich możliwych terminali. Powinieneś zrozumieć, że powyższy listing nie jest dosłownym przełożeniem tego, jak będą wyglądać Twoje dane wyjściowe. Musisz natomiast koncentrować się na tekście wyświetlanym między pierwszą linią `$ python3.6...` i ostatnią linią ze znakiem `$`.

Zrób to sam

1. Sprawdź, czy dobrze rozumiesz przeznaczenie znaku `#` i upewnij się, że znasz różne jego nazwy (na przykład płotek czy hash).
2. Przeanalizuj każdą linię kodu z pliku `ex2.py` wstecz. Zacznij od ostatniej linii i sprawdzaj każde słowo, które powinieneś był wpisać.

3. Czy znalazłeś więcej błędów? Usuń je.
4. Przeczytaj głośno to, co wpisałeś powyżej, wymawiając przy tym pełną nazwę każdego znaku specjalnego. Znalazłeś więcej błędów? Usuń je.

Typowe pytania

Czy na pewno znak # nazywany jest również hashem? Nazywam ten znak kratką, ponieważ jest to nazwa rozpoznawalna w każdym kraju. Każdy kraj uważa swoją nazwę dla tego znaku za najważniejszą i najtrafniejszą. Dla mnie jest to po prostu arogancja. Trzeba wyluzować i skoncentrować się na ważniejszych sprawach, takich jak nauka kodowania.

Dlaczego znak # w poleceniu `print("Hej # tam.")` nie jest ignorowany? Znak # w tym kodzie znajduje się wewnątrz łańcucha znaków i dlatego będzie stanowił część tego łańcucha, który kończy się na znaku ". Znaki kratki w łańcuchach są uznawane za zwykłe znaki, a nie komentarze.

Jak wykomentować wiele linii? Umieść znak # przed każdą z nich.

Nie wiem, jak wpisać znak # na mojej klawiaturze, która ma inny układ językowy, charakterystyczny dla mojego kraju. Jak mogę to zrobić? W niektórych krajach do wpisywania znaków obcych dla danego języka używa się kombinacji przycisku *Alt* lub *Shift* z innymi przyciskami klawiatury. Informacji o sposobie wpisywania znaku # na Twojej klawiaturze będziesz musiał poszukać w internecie.

Dlaczego muszę czytać kod wstecz? Dzięki tej sztuczce Twój mózg nie będzie przywiązywał się do znaczenia poszczególnych części kodu i będziesz mógł przetworzyć każdy fragment bardzo dokładnie. Jest to przydatna technika sprawdzania błędów.

Liczby i działania algebraiczne

Każdy język programowania ma własny sposób radzenia sobie z liczbami i działaniami algebraicznymi. Nie martw się: programiści często kłamią, że są geniuszami matematycznymi, podczas gdy naprawdę nie są. Gdyby nimi byli, zajmowaliby się matematyką, a nie pisanie dziwacznych frameworków webowych, aby zarobić na sportowe auta.

To ćwiczenie zawiera wiele symboli matematycznych. Nazwijmy je od razu, żebyś wiedział, o czym mowa. Podczas wpisywania poszczególnych symboli wymawiaj ich pełne nazwy. Kiedy Ci się to znudzi, możesz przestać. Oto nazwy symboli:

- `+`: plus,
- `-`: minus,
- `/`: ukośnik,
- `*`: gwiazdka,
- `%`: procent,
- `<`: mniejsze niż,
- `>`: większe niż,
- `<=`: mniejsze lub równe,
- `>=`: większe lub równe.

Zwróciłeś uwagę, że brakuje operacji arytmetycznych? Po wpisaniu kodu tego ćwiczenia wróć do powyższego zestawienia, zastanów się, jakie są funkcje poszczególnych symboli, i uzupełnij tabelę, na przykład znak `+` to dodawanie.

ex3.py

```
1  print("Policzę teraz pogłowie mojego drobiu:")
2
3  print("Kury", 25 + 30 / 6)
4  print("Koguty", 100 - 25 * 3 % 4)
5
6  print("Teraz policzę jaja:")
7
8  print(3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6)
9
10 print("Czy to prawda, że 3 + 2 < 5 - 7?")
11
12 print(3 + 2 < 5 - 7)
13
14 print("Ile to jest 3 + 2?", 3 + 2)
15 print("Ile to jest 5 - 7?", 5 - 7)
16
17 print("Oj, to dlatego to daje False.")
18
```

```
19 print("Poćwiczmy jeszcze trochę.")
20
21 print("Czy to jest większe?", 5 > -2)
22 print("Czy to jest większe lub równe?", 5 >= -2)
23 print("Czy to jest mniejsze lub równe?", 5 <= -2)
```

Zanim uruchomisz plik, upewnij się, że wpisałeś dokładnie to, co wypisałem w listingu. Porównaj każdą linię swojego pliku z moim.

Co powinieneś zobaczyć

Ćwiczenie 3. — sesja

```
$ python3.6 ex3.py
Policzę teraz pogłowie mojego drobiu:
Kury 30.0
Koguty 97
Teraz policzę jaja:
6.75
Czy to prawda, że  $3 + 2 < 5 - 7$ ?
False
Ile to jest  $3 + 2$ ? 5
Ile to jest  $5 - 7$ ? -2
Oj, to dlatego to daje False.
Poćwiczmy jeszcze trochę.
Czy to jest większe? True
Czy to jest większe lub równe? True
Czy to jest mniejsze lub równe? False
```

Zrób to sam

1. Nad każdą linią kodu użyj znaku `#`, aby napisać komentarz wyjaśniający, co robi.
2. Pamiętasz z ćwiczenia 0, jak uruchamiałeś `python3.6`? Ponownie uruchom polecenie `python3.6` w ten sam sposób i za pomocą operatorów matematycznych użyj Pythona jako kalkulatora.
3. Znajdź coś, co musisz obliczyć, i napisz nowy plik `.py`, który to robi.
4. Napisz ćwiczenie `ex3.py` od nowa, używając liczb zmiennoprzecinkowych, aby obliczenia były bardziej dokładne. Liczbą zmiennoprzecinkową jest na przykład `20.0`.

Typowe pytania

Dlaczego znak `%` to operacja modulo, a nie procent? Głównie dlatego, że takie przeznaczenie dla tego symbolu wybrali projektanci. W normalnym zapisie matematycznym prawidłowo odczytalibyśmy to oczywiście jako procent. Jednak w programowaniu obliczenia procentowe są zazwyczaj wykonywane przy użyciu prostego dzielenia i operatora `/`. Modulo jest inną operacją, która po prostu używa symbolu `%`.

Jak działa modulo (%)? Inaczej powiedzielibyśmy, że „ X podzielone przez Y daje resztę J ”, na przykład „100 podzielone przez 16 daje resztę 4”. Wynik operacji modulo to wartość J , czyli reszta z dzielenia.

Jaka jest kolejność wykonywania działań? W języku angielskim do określenia kolejności wykonywania działań używa się akronimu **PEMDAS** (ang. *Parentheses Exponents Multiplication Division Addition Subtraction*), co oznacza: nawiasy, potęgowanie, mnożenie, dzielenie, dodawanie, odejmowanie. Ta kolejność jest również przestrzegana w Pythonie. Częstym błędem jest potraktowanie wskazówki PEMDAS jako ścisłej kolejności, czyli „Najpierw P (nawiasy), potem E (potęgowanie), potem M (mnożenie), potem D (dzielenie), potem A (dodawanie) i na koniec S (odejmowanie)”. Właściwą kolejnością jest wykonywanie mnożenia i dzielenia ($M\&D$) w jednym kroku, od lewej do prawej, a następnie dodawania i odejmowania ($A\&S$) w jednym kroku, od lewej do prawej. Możemy więc zapisać PEMDAS jako $PE(M\&D)(A\&S)$.

Zmienne i nazwy

Potrafiś już drukować różne rzeczy za pomocą polecenia `print` oraz wykonywać obliczenia matematyczne. Kolejnym krokiem jest poznanie zmiennych. W programowaniu zmienna to nic innego jak nazwa dla czegoś, tak jak „Zed” to nazwa „osoby, która napisała tę książkę”. Programiści używają nazw zmiennych, aby kod był bardziej czytelny dla człowieka i dlatego, że mają kiepską pamięć. Gdyby nie używali dobrych nazw dla różnych elementów oprogramowania, pogubiliby się przy próbie ponownego odczytania własnego kodu.

Jeśli utkniesz na którymś etapie tego ćwiczenia, pamiętaj o poznanych do tej pory sztuczkach, które pozwalają wyszukiwać różnice i skupiać się na szczegółach.

1. Nad każdą linią kodu napisz komentarz wyjaśniający, co robi.
2. Przeczytaj swój plik `.py` wstecz.
3. Przeczytaj swój plik `.py` na głos, wymawiając nawet pełne nazwy znaków specjalnych.

ex4.py

```
1 cars = 100
2 space_in_a_car = 4.0
3 drivers = 30
4 passengers = 90
5 cars_not_driven = cars - drivers
6 cars_driven = drivers
7 carpool_capacity = cars_driven * space_in_a_car
8 average_passengers_per_car = passengers / cars_driven
9
10
11 print("Dostępnych jest", cars, "samochodów.")
12 print("Dostępnych jest tylko", drivers, "kierowców.")
13 print("Dziś będzie", cars_not_driven, "pustych samochodów.")
14 print("Dziś możemy przetransportować", carpool_capacity, "osób.")
15 print("Mamy dziś do przewiezienia", passengers, "pasażerów.")
16 print("Musimy umieścić średnio", average_passengers_per_car,
17       "osoby w każdym samochodzie.")
```

OSTRZEŻENIE! Znak `_` w zmiennej `space_in_a_car` jest nazywany **znakem podkreślenia** (podkreślnikiem). Dowiedz się, jak wpisywać ten znak, jeśli jeszcze tego nie potrafisz. Używamy tego znaku dość często, aby umieszczać wymagowane spacje między słowami w nazwach zmiennych.

Co powinieneś zobaczyć

Ćwiczenie 4. — sesja

```
$ python3.6 ex4.py
Dostępnych jest 100 samochodów.
Dostępnych jest tylko 30 kierowców.
Dziś będzie 70 pustych samochodów.
Dziś możemy przetransportować 120.0 osób.
Mamy dziś do przewiezienia 90 pasażerów.
Musimy umieścić średnio 3.0 osoby w każdym samochodzie.
```

Zrób to sam

Kiedy pisałem ten program po raz pierwszy, zrobiłem błąd, a Python powiedział mi o tym w ten sposób:

```
Traceback (most recent call last):
  File "ex4.py", line 8, in <module>
    average_passengers_per_car = car_pool_capacity / passenger
NameError: name 'car_pool_capacity' is not defined
```

Wytłumacz ten błąd własnymi słowami. Pamiętaj o używaniu numerów linii i wyjaśnij, dlaczego należy tak robić.

Oto kilka kolejnych ćwiczeń.

1. Dla zmiennej `space_in_a_car` użyłem liczby `4.0`, ale czy jest to konieczne? Co się stanie, jeśli wpiszesz po prostu `4`?
2. Pamiętaj, że `4.0` to **liczba zmiennoprzecinkowa**, czyli po prostu liczba z kropką dziesiętną. Aby uzyskać liczbę zmiennoprzecinkową, powinieneś wpisać `4.0` zamiast `4`.
3. Napisz komentarz nad każdym przypisaniem zmiennej.
4. Upewnij się, że znasz nazwę znaku `=` (znaku równości) i jego zastosowanie, czyli nadawanie nazw (`cars_driven`, `passengers`) danym (liczbom, łańcuchom znaków i podobnym).
5. Pamiętaj, że znak `_` jest znakiem podkreślenia.
6. Spróbuj, tak jak przedtem, uruchomić z terminala `python3.6` jako kalkulator i użyć nazw zmiennych do wykonywania obliczeń. Popularne nazwy zmiennych to `i`, `x` i `j`.

Typowe pytania

Jaka jest różnica między = (pojedynczym znakiem równości) a == (podwójnym znakiem równości)? Pojedynczy znak równości (=) przypisuje wartość z prawej do zmiennej po lewej stronie. Podwójny znak równości (==) sprawdza, czy dwie rzeczy mają tę samą wartość. Dowiesz się o tym w ćwiczeniu 27.

Czy możemy napisać `x=100` zamiast `x = 100`? Możesz tak zrobić, ale jest to zła forma. Powinieneś dodawać spacje wokół takich operatorów, aby kod był bardziej czytelny.

Co rozumiesz przez „przeczytaj plik wstecz”? To bardzo proste. Wyobraź sobie, że masz plik z 16 liniami kodu. Zaczynij od linii 16. i porównaj ją z odpowiednią linią w moim pliku. Potem zrób to samo dla linii 15. i kolejnych, dopóki nie przeczytasz całego pliku wstecz.

Dlaczego użyłeś liczby 4.0 dla zmiennej `space_in_a_car`? Głównie dlatego, żebyś dowiedział się, czym jest liczba zmiennoprzecinkowa, i mógł zadać to pytanie. Zobacz podrozdział „Zrób to sam”.

Więcej zmiennych i drukowania

Teraz będziemy wpisywać i drukować jeszcze więcej zmiennych. Tym razem użyjemy tak zwanego **sformatowanego łańcucha znaków** (ang. *format string*). Za każdym razem, gdy umieszczasz fragment tekstu między podwójnymi cudzysłowami (" "), tworzysz **łańcuch znaków** (ang. *string*). Jest on czymś, co Twój program może przedstawić człowiekowi. Łańcuchy znaków można między innymi drukować, zapisywać w plikach lub wysyłać do serwerów WWW.

Łańcuchy znaków są bardzo przydatne, dlatego w tym ćwiczeniu dowiesz się, jak tworzyć łańcuchy znaków z osadzonymi zmiennymi. Zmienną osadza się w łańcuchu znaków za pomocą specjalnej sekwencji nawiasów klamrowych {}. Taki łańcuch znaków należy również rozpocząć literą f (od słowa „format”), na przykład f"Witaj, {jakaś_zmienna}". Litera f przed znakiem podwójnego cudzysłowu (") i znaki {} przekazują Pythonowi 3 następującą informację: „Hej, ten łańcuch znaków musi być sformatowany. Umieść w nim te zmienne”.

Jak zwykle, wpisz poniższy kod, nawet jeśli go nie zrozumiesz. Wpisz go dokładnie tak samo.

ex5.py

```

1  my_name = 'Zed A. Shaw'
2  my_age = 35 # nie kłamię
3  my_height = 190 # centymetrów
4  my_weight = 80 # kilogramów
5  my_eyes = 'Niebieskie'
6  my_teeth = 'Białe'
7  my_hair = 'Brązowe'
8
9  print(f"Porozmawiajmy o {my_name}.")
10 print(f"Ma {my_height} centymetrów wzrostu.")
11 print(f"Waży {my_weight} kilogramów.")
12 print("Tak naprawdę nie waży dużo.")
13 print(f"Ma {my_eyes} oczy i {my_hair} włosy.")
14 print(f"Jego zęby są zazwyczaj {my_teeth}, w zależności od ilości wypitej kawy.")
15
16 # ta linia jest trudna, więc wpisz ją uważnie
17 total = my_age + my_height + my_weight
18 print(f"Jeśli dodam {my_age}, {my_height} i {my_weight}, otrzymam {total}.")

```

Co powinieneś zobaczyć

Ćwiczenie 5. — sesja

```

$ python3.6 ex5.py
Porozmawiajmy o Zed A. Shaw.
Ma 190 centymetrów wzrostu.

```

Waży 80 kilogramów.
Tak naprawdę nie waży dużo.
Ma Niebieskie oczy i Brązowe włosy.
Jego zęby są zazwyczaj Białe, w zależności od ilości wypitej kawy.
Jeśli dodam 35, 190 i 80, otrzymam 305.

Zrób to sam

1. Zmodyfikuj wszystkie zmienne, usuwając z każdej przedrostek `my_`. Upewnij się, że zmieniłeś nazwy we wszystkich wystąpieniach, a nie tylko w miejscach przypisywania zmiennym wartości za pomocą znaku `=`.
2. Spróbuj napisać kilka zmiennych, które konwertują centymetry i kilogramy na cale i funty. Nie wpisuj po prostu nowych miar. Zastosuj działania algebraiczne w Pythonie.

Typowe pytania

Czy mogę utworzyć zmienną w następujący sposób: `1 = 'Zed Shaw'`? Nie, `1` nie jest poprawną nazwą zmiennej. Zmienne muszą zaczynać się od znaku, więc `a1` zadziała, ale `1` nie.

W jaki sposób mogę zaokrąglić liczbę zmiennoprzecinkową? Możesz użyć funkcji `round()` w następujący sposób: `round(1.7333)`.

Dlaczego nic z tego nie rozumiem? Spróbuj potraktować liczby w tym skrypcie jak własne miary. Może to dziwne, ale mówienie o sobie sprawi, że będzie się to wydawać bardziej realne. Poza tym dopiero zaczynasz, więc nie wszystko od razu będzie miało sens. Wykonuj kolejne ćwiczenia, a z każdym krokiem będziesz rozumiał więcej.

Łańcuchy znaków i tekst

Choć pisałeś już łańcuchy znaków, wciąż nie wiesz, co one robią. W tym ćwiczeniu utworzymy kilka zmiennych ze złożonymi łańcuchami znaków, abyś mógł zobaczyć, do czego służą. Najpierw wyjaśnię, czym jest łańcuch znaków.

Łańcuch znaków (ang. *string*) to zwykle tekst, który chcemy komuś wyświetlić lub „wyeksportować” z pisanego programu. Python „wie”, że coś ma być łańcuchem znaków, gdy tekst umieszczamy w podwójnych (" ") lub pojedynczych cudzysłowach (' ') . Widziałeś to wiele razy, gdy po poleceniu `print` umieszczałeś tekst między znakami ' lub ", aby wydrukować łańcuch znaków.

Łańcuchy znaków mogą zawierać dowolną liczbę zmiennych, które znajdują się w Twoim skrypcie Pythona. Pamiętaj, że zmienną jest każda linia kodu, w której ustawisz jakąś nazwę równą (=) pewnej wartości. W kodzie tego ćwiczenia `types_of_people = 10` tworzy zmienną o nazwie `types_of_people` (rodzaje ludzi) i ustawia ją równą (=) wartości 10. Możesz umieścić tę wartość w dowolnym łańcuchu znaków za pomocą zapisu `{types_of_people}`. Widać również, że trzeba użyć specjalnego typu łańcucha znaków w celu „sformatowania”; nazywa się to *f-string* (jest to sformatowany literał łańcucha znaków) i wygląda następująco:

```
f"tu jakiś tekst {jakaś_zmienna}"  
f"jakiś inny tekst {kolejna_zmienna}"
```

Python oferuje również inny rodzaj formatowania przy użyciu składni `.format()`, którą widzisz w linii 17. Będę używał tej składni czasami, gdy będę chciał zastosować formatowanie do już utworzonego łańcucha znaków, na przykład w pętli. Omówię to szczegółowo później.

Wpiszemy teraz całą masę łańcuchów znaków, zmiennych i formatów, a następnie wydrukujemy je. Poćwiczmy także posługiwanie się skróconymi nazwami zmiennych. Programiści uwielbiają oszczędzać czas Twoim kosztem, używając denerwująco krótkich i tajemniczych nazwy zmiennych, więc od razu zacznijmy je czytać i pisać.

ex6.py

```
1  types_of_people = 10  
2  x = f"Istnieje {types_of_people} rodzajów ludzi."  
3  
4  binary = "binarny"  
5  do_not = "nie znają"  
6  y = f"Ci, którzy znają system {binary} i ci, którzy go {do_not}."  
7  
8  print(x)  
9  print(y)  
10  
11 print(f"Powiedziałem: {x}")  
12 print(f"Powiedziałem również: '{y}'")  
13  
14 hilarious = False
```

```
15 joke_evaluation = "Czyż to nie jest przeżabawny dowcip?! {}"  
16  
17 print(joke_evaluation.format(hilarious))  
18  
19 w = "To jest lewa strona..."  
20 e = "łańcucha znaków z prawą stroną."  
21  
22 print(w + e)
```

Co powinieneś zobaczyć

Ćwiczenie 6. — sesja

```
$ python3.6 ex6.py  
Istnieje 10 rodzajów ludzi.  
Ci, którzy znają system binarny i ci, którzy go nie znają.  
Powiedziałem: Istnieje 10 rodzajów ludzi.  
Powiedziałem również: 'Ci, którzy znają system binarny i ci, którzy go nie znają.'  
Czyż to nie jest przeżabawny dowcip?! False  
To jest lewa strona...łańcucha znaków z prawą stroną.
```

Zrób to sam

1. Przeanalizuj cały program i nad każdą linią napisz komentarz wyjaśniający.
2. Znajdź wszystkie miejsca, w których łańcuch znaków jest umieszczony w innym łańcuchu znaków. Istnieje cztery takie miejsca.
3. Jesteś pewien, że są tylko cztery miejsca? Skąd wiesz? Może lubię kłamać.
4. Wyjaśnij, dlaczego dodanie dwóch łańcuchów znaków w i e za pomocą operatora + tworzy dłuższy łańcuch znaków.

Popsuj kod

Jesteś teraz w punkcie, w którym możesz spróbować popsuć kod, aby zobaczyć, co się stanie. Potraktuj to jako grę polegającą na wymyśleniu najsprytniejszego sposobu popsucia kodu. Możesz również poszukać najprostszego sposobu, aby popsuć kod. Gdy już to zrobisz, będziesz musiał go naprawić. Jeśli masz znajomego, możecie popracować w parach, psując i naprawiając wzajemnie swoje kody. Przekaż swojemu znajomemu plik `ex6.py`, aby mógł coś zepsuć. Potem Ty spróbuj znaleźć jego błąd i naprawić go. Baw się dobrze i pamiętaj, że jeśli napisałeś ten kod raz, możesz zrobić to jeszcze raz. Jeżeli za bardzo popsujesz kod, zawsze możesz go wpisać ponownie w ramach dodatkowego ćwiczenia.

Typowe pytania

Dlaczego umieszczasz niektóre łańcuchy znaków w pojedynczych cudzysłowach (' '), a inne w podwójnych (" ")? Głównie ze względu na styl, ale używam pojedynczych cudzysłowów wewnątrz łańcucha znaków ujętego w podwójne cudzysłowy. Spójrz na linie 6. i 15., aby zobaczyć, jak to robię.

Jeśli uznałbyś ten żart za zabawny, to mógłbyś napisać `hilarious = True`? Tak. Więcej na temat tych wartości logicznych dowiesz się w ćwiczeniu 27.

Więcej drukowania

Teraz wykonasz kilka ćwiczeń, w których będziesz po prostu wpisywał kod i uruchamiał go. Nie będę wyjaśniał tego ćwiczenia, ponieważ jest podobne do poprzednich. Celem jest „wyrobienie mięśni”. Do zobaczenia za kilka ćwiczeń i *niczego nie pomijaj! Nie przeklejjaj!*

ex7.py

```

1  print("Mary małą owcę ma.")
2  print("Która biała jest jak {}".format('mgła'))
3  print("Raz do szkoły przyszły dwie.")
4  print("." * 10) # co to będzie robić?
5
6  end1 = "C"
7  end2 = "h"
8  end3 = "e"
9  end4 = "e"
10 end5 = "s"
11 end6 = "e"
12 end7 = "B"
13 end8 = "u"
14 end9 = "r"
15 end10 = "g"
16 end11 = "e"
17 end12 = "r"
18
19 # zwróć uwagę na przecinek na końcu; spróbuj go usunąć, żeby zobaczyć, co się stanie
20 print(end1 + end2 + end3 + end4 + end5 + end6, end=' ')
21 print(end7 + end8 + end9 + end10 + end11 + end12)

```

Co powinieneś zobaczyć

Ćwiczenie 7. — sesja

```

$ python3.6 ex7.py
Mary małą owcę ma.
Która biała jest jak mgła.
Raz do szkoły przyszły dwie.
.....
Cheese Burger

```

Zrób to sam

Dla kilku kolejnych ćwiczeń będziesz miał do wykonania dokładnie te same ćwiczenia samodzielne.

1. Wróć do kodu i napisz komentarz na temat tego, co robi każda linia.
2. Odczytaj każdą linię kodu wstecz lub na głos, aby znaleźć błędy.
3. Od tej pory, kiedy popełnisz jakiś błąd, zapisz na kartce papieru, jaki rodzaj błędu popełniłeś.
4. Kiedy przejdiesz do następnego ćwiczenia, popatrz na błędy, które popełniłeś w tym ćwiczeniu, i postaraj się ich nie powtórzyć.
5. Pamiętaj, że każdy popełnia błędy. Programiści to magicy, którzy nabierają wszystkich, że są doskonali i nigdy się nie mylą, ale to tylko udawanie. Popełniają błędy cały czas.

Popsuj kod

Dobrze się bawiłeś, psując kod w ćwiczeniu 6.? Od tej pory będziesz psuł każdy kod, który napiszesz, lub kod znajomego. Nie w każdym ćwiczeniu będę zamieszczał podrozdział „Popsuj kod”, ale Twoim celem jest znalezienie tak wielu różnych sposobów na popsucie kodu, aż się nie zmęczysz lub nie wyczerpiesz wszystkich możliwości. W niektórych ćwiczeniach mogę wskazać konkretny, powszechny sposób na popsucie kodu, ale we wszystkich przypadkach pamiętaj, że jest to Twoje stałe zadanie.

Typowe pytania

Dlaczego używasz zmiennej o nazwie 'mgła'? To właściwie nie jest zmienna. Jest to po prostu łańcuch znaków, w którym umieszczone zostało słowo mgła. Zmienna nie byłaby ujęta w pojedyncze cudzysłowy.

Czy normalne jest pisanie komentarza do każdej linii kodu, tak jak polecasz w ćwiczeniu 1., w podrozdziale „Zrób to sam”? Nie. Komentarze pisze się tylko po to, aby wyjaśnić trudny do zrozumienia kod lub uzasadnić, dlaczego coś się zrobiło. To uzasadnienie jest zazwyczaj o wiele ważniejsze i na jego podstawie próbuje się napisać kod, który sam wyjaśnia, w jaki sposób dana rzecz jest wykonywana. Czasami jednak w celu rozwiązania jakiegoś problemu trzeba napisać tak nieprzyjemny kod, że wymaga on komentarza w każdej linii. W naszym przypadku ma to na celu wyłącznie przećwiczenie tłumaczenia kodu na język polski.

Czy do tworzenia łańcuchów znaków mogę używać pojedynczych i podwójnych cudzysłowów zamiennie, czy też mają one różne zastosowania? W języku Python dopuszczalne są oba te sposoby tworzenia łańcuchów znaków, chociaż zazwyczaj pojedynczych cudzysłowów używa się do krótkich łańcuchów, takich jak 'a' lub 'mgła'. ")?

Teraz zobaczysz, jak wykonać bardziej skomplikowane formatowanie łańcucha znaków. Ten kod wygląda na złożony, ale jeśli napiszesz komentarze nad każdą linią i rozłożysz poszczególne rzeczy na części, zrozumiesz go.

```

1  formatter = "{} {} {} {}"
2
3  print(formatter.format(1, 2, 3, 4))
4  print(formatter.format("jeden", "dwa", "trzy", "cztery"))
5  print(formatter.format(True, False, False, True))
6  print(formatter.format(formatter, formatter, formatter, formatter))
7  print(formatter.format(
8      "Spróbuj tutaj",
9      "Wpisać własny tekst",
10     "Może wiersz",
11     "Albo piosenkę o lęku"
12 ))

```

Ćwiczenie 8. — sesja

```
$ python3.6 ex8.py  
1 2 3 4  
jeden dwa trzy cztery  
True False False True  
{ } { } { } { } { } { } { } { } { } { } { } { } { }  
Spróbuj tutaj wpisać własny tekst Może wiersz Albo piosenke o leku
```

W tym ćwiczeniu używam tak zwanej **funkcji**, która przekształca zmienną `formatter` w inne łańcuchy znaków. Kiedy piszę `formatter.format(...)`, mówię Pythonowi, aby wykonał następujące czynności.

1. Pobrał łańcuch znaków formatter zdefiniowany w linii 1.
2. Wywołał jego funkcję format, co daje efekt podobny do wykonania w wierszu poleceń polecenia o nazwie format.
3. Przekazał do funkcji format cztery argumenty, które odpowiadają czterem sekwencjom znaków {} w zmiennej formatter. Jest to podobne do przekazywania argumentów do polecenia format w wierszu poleceń.
4. Wynikiem wywołania funkcji format na zmiennej formatter jest nowy łańcuch znaków, w którym sekwencje {} zostały zastąpione przez cztery zmienne. Właśnie to drukuje teraz polecenie print.

To dużo jak na jedno ćwiczenie, więc chcę, żebyś potraktował to jako łamigłówkę. Nie ma problemu, jeśli *nie całkiem rozumiesz*, co się tutaj wyprawia, ponieważ reszta książki powoli to wyjaśni. Na tym etapie postaraj się po prostu przestudiować ten kod i zobaczyć, co się w nim dzieje, a potem przejdź do następnego ćwiczenia.

Zrób to sam

Sprawdź poprawność wpisanego kodu, zapisz swoje błędy i postaraj się nie popełniać tych samych błędów w następnym ćwiczeniu. Innymi słowy, powtórz „Zrób to sam” z ćwiczenia 7.

Typowe pytania

Dlaczego muszę używać cudzysłowów wokół "jeden", ale nie wokół True lub False? Python rozpoznaje True i False jako słowa kluczowe reprezentujące pojęcie prawdy i fałszu. Jeśli umieścisz je w cudzysłowie, zostaną zmienione w łańcuchy znaków i nie będą działać. Więcej na temat działania tych słów kluczowych dowiesz się w ćwiczeniu 27.

Czy do uruchomienia tego kodu mogę użyć IDLE? Nie, powinieneś nauczyć się korzystać z wiersza poleceń. Jest to ważna kwestia podczas nauki programowania i dobry początek, jeśli chcesz dowiedzieć się czegoś o programowaniu. W dalszych partiach książki i tak nie będziesz w stanie skorzystać z IDLE.

Drukowanie, drukowanie, drukowanie

Prawdopodobnie zorientowałeś się już, że w tej książce używam więcej niż jednego ćwiczenia, aby nauczyć Cię czegoś nowego. Zaczynam od kodu, którego możesz nie rozumieć, a potem kolejne ćwiczenia wyjaśniają daną koncepcję. Jeśli teraz czegoś nie rozumiesz, zrozumiesz to później, gdy wykonasz więcej ćwiczeń. Zapisuj, czego nie rozumiesz, i idź dalej.

ex9.py

```

1  # Tu jest trzecie nowych, dziwnych rzeczy; pamiętaj, żeby wpisać wszystko dokładnie
2
3  days = "Pon Wt Śr Czw Pt Sob Niedz"
4  months = "Sty\nLut\nMar\nKwi\nMaj\nCze\nLip\nSie"
5
6  print("Oto dni tygodnia: ", days)
7  print("Oto miesiące: ", months)
8
9  print("""
10 Tutaj coś się dzieje.
11 Używamy trzech podwójnych cudzysłowów.
12 Będziemy w stanie wpisać dowolnie dużo tekstu.
13 Jeśli chcemy, nawet 4 linie albo 5 lub 6.
14 """)

```

Co powinieneś zobaczyć

Ćwiczenie 9. — sesja

```

$ python3.6 ex9.py
Oto dni tygodnia: Pon Wt Śr Czw Pt Sob Niedz
Oto miesiące: Sty
Lut
Mar
Kwi
Maj
Cze
Lip
Sie

```

```

Tutaj coś się dzieje.
Używamy trzech podwójnych cudzysłowów.
Będziemy w stanie wpisać dowolnie dużo tekstu.
Jeśli chcemy, nawet 4 linie albo 5 lub 6.

```

Zrób to sam

Sprawdź swoją pracę, zapisz błędy i spróbuj nie popełniać ich przy następnym ćwiczeniu. Psujesz swój kod, a następnie go naprawiasz? Innymi słowy, powtórz „Zrób to sam” z ćwiczenia 7.

Typowe pytania

Dlaczego pojawia się błąd, gdy umieszczam spację między trzema podwójnymi cudzysłowami? Musisz wpisać je tak: `" "`, a nie tak: `" " "`, czyli **bez spacji** między poszczególnymi cudzysłowami.

Co zrobić, jeśli chciałbym rozpocząć wymienianie miesięcy w nowej linii? Po prostu rozpocznij łańcuch znaków od sekwencji `\n`, tak jak poniżej:

```
"\nSty\nLut\nMar\nKwi\nMaj\nCze\nLip\nSie"
```

Czy to źle, że moje błędy są zawsze błędami w pisowni? Na początku (a nawet później) większość błędów programowania to proste błędy ortograficzne, literówki lub psucie prostych rzeczy.

Co to było?

W ćwiczeniu 9. wprowadziłem trochę nowych rzeczy, abyś cały czas zachowywał czujność. Pokazałem dwa sposoby tworzenia łańcucha znaków, który rozciąga się na wiele linii. W pierwszej metodzie umieściłem sekwencję znaków `\n` (lewy ukośnik, n) pomiędzy poszczególnymi nazwami miesięcy. Ta sekwencja wstawia w danym miejscu łańcucha znaków znak nowej linii.

Znak lewego ukośnika (`\`) koduje w łańcuchu znaków trudne do wpisania znaki. Istnieje wiele **sekwencji ucieczki** (ang. *escape sequences*) dla różnych znaków, z których możesz skorzystać. Wypróbujemy kilka z tych sekwencji, żebyś zobaczył, co mam na myśli.

Ważną sekwencją ucieczki jest wstawianie pojedynczego (`'`) lub podwójnego cudzysłowu (`"`)?. Wyobraź sobie, że masz łańcuch znaków, w którym użyto podwójnych cudzysłowów, i chcesz umieścić w nim podwójny cudzysłów. Jeśli napiszesz `"Ja "rozumiem" Jasia."`, wtedy Python „pogubi się”, ponieważ pomyśli, że podwójne cudzysłowy wokół `"rozumiem"` w rzeczywistości *kończą* łańcuch znaków. Potrzebujesz sposobu, aby powiedzieć Pythonowi, że podwójny cudzysłów w środku łańcucha znaków nie jest *prawdziwym* podwójnym cudzysłowem.

Aby rozwiązać ten problem, stosuje się **sekwencję ucieczki** dla podwójnych i pojedynczych cudzysłowów, aby Python „wiedział”, że powinien je wstawić do łańcucha znaków. Oto przykład.

```
"Mam 6'2\" wzrostu." # wstawiamy podwójny cudzysłów do łańcucha znaków
'Mam 6\'2" wzrostu.' # wstawiamy pojedynczy cudzysłów do łańcucha znaków
```

Druga metoda tworzenia łańcucha znaków polega na użyciu trzech podwójnych cudzysłowów (`"""`), co również działa jak łańcuch znaków, ale przed zamykającą sekwencją cudzysłowów (`"""`) można wpisać dowolną liczbę linii tekstu. Tym również się pobawimy.

ex10.py

```
1 tabby_cat = "\tJestem tabulatorem."
2 persian_cat = "Jestem podziałem\nlinii."
3 backslash_cat = "Jestem sobie \\ jakimś \\ kotem."
4
5 fat_cat = """
6 Zrobię listę:
7 \t* Jedzenie dla kotka
8 \t* Rybki
9 \t* Kocimiętka\n\t* Trawa
10 """
11
12 print(tabby_cat)
13 print(persian_cat)
14 print(backslash_cat)
15 print(fat_cat)
```


Co powinienś zobaczyć

Poszukaj znaków tabulacji, które wstawiłeś. W tym ćwiczeniu ważne jest, aby odstępy były prawidłowe.

```
$ python ex10.py
    Jestem tabulatorem.
Jestem podziałem
linii.
Jestem sobie \ jakimś \ kotem.
```

```
Zrobię listę:
* Jedzenie dla kotka
* Rybki
* Kocimiętka
* Trawa
```

Sekwencje ucieczki

W poniższej tabeli przedstawione zostały wszystkie sekwencje ucieczki obsługiwane przez Pythona. Możesz nie używać wielu z nich, ale i tak powinienś zapamiętać ich format i funkcję. Wypróbuj je w łańcuchach znaków, aby sprawdzić, czy potrafisz je zastosować.

Sekwencja ucieczki	Funkcja
\\	Lewy ukośnik (\)
\'	Pojedynczy cudzysłów (')
\"	Podwójny cudzysłów (")
\a	Znak dzwonka ASCII (BEL)
\b	Znak backspace ASCII (BS)
\f	Znak wysunięcia strony ASCII (FF)
\n	Znak nowej linii (LF)
\N{ <i>nazwa</i> }	Znak o nazwie <i>nazwa</i> z bazy danych Unicode (tylko Unicode)
\r	Powrót karetki (CR)
\t	Tabulator poziomy (TAB)
\uxxxx	Znak reprezentowany przez 16-bitową wartość szesnastkową <i>xxxx</i>
\Uxxxxxxxx	Znak reprezentowany przez 32-bitową wartość szesnastkową <i>xxxxxxxx</i>
\v	Tabulator pionowy ASCII (VT)
\ooo	Znak reprezentowany przez ósemkową wartość <i>ooo</i>
\xhh	Znak reprezentowany przez szesnastkową wartość <i>hh</i>

Zrób to sam

1. Zapamiętaj wszystkie sekwencje ucieczki, zapisując je na fiszkach.
2. Zamiast trzech podwójnych cudzysłowów (""") użyj trzech pojedynczych (''''). Czy widzisz, dlaczego jest to możliwe?
3. Połącz sekwencje ucieczki i sformatowane łańcuchy znaków, aby utworzyć bardziej złożone formatowanie.

Typowe pytania

Wciąż nie do końca zrozumiałem ostatnie ćwiczenie. Czy powinienem kontynuować? Tak, idź dalej. Zamiast się zatrzymywać, zapisuj wszystkie rzeczy, których nie rozumiesz w poszczególnych ćwiczeniach. Co pewien czas przejrzyj swoje notatki i zobacz, czy po wykonaniu większej liczby ćwiczeń już rozumiesz niektóre rzeczy. Czasami będziesz musiał cofnąć o kilka ćwiczeń i wykonać je ponownie.

Co sprawia, że sekwencja znaków \ jest w porównaniu do innych sekwencją specjalną? Jest to po prostu sposób zapisywania pojedynczego znaku lewego ukośnika (\). Zastanów się, dlaczego możesz tego potrzebować.

Kiedy piszę // lub /n, to nie działa\. Dzieje się tak, ponieważ używasz prawego ukośnika (/) zamiast lewego (\). Są to dwa różne znaki z całkowicie odmiennym przeznaczeniem.

Nie rozumiem ćwiczenia 3. z podrozdziału „Zrób to sam”. Co rozumiesz przez „połączenie” sekwencji ucieczki i sformatowanych łańcuchów znaków? Jedną z koncepcji, którą powinieneś zrozumieć, jest to, że poszczególne ćwiczenia można łączyć ze sobą w celu rozwiązania określonych problemów. Przypomnij sobie, co wiesz o sformatowanych łańcuchach znaków, i napisz jakiś nowy kod, który używa takich łańcuchów oraz sekwencji ucieczki z tego ćwiczenia.

Co jest lepsze, ''' czy """""? Jest to całkowicie zależne od stylu. Na razie stosuj styl trzech pojedynczych cudzysłowów (''''), ale bądź gotowy użyć również drugiego stylu, w zależności od tego, co wydaje się najlepsze lub co robią wszyscy pozostali.

Zadawanie pytań

Nadszedł czas, aby zwiększyć tempo. Dużo drukowałeś, by zapoznać się z pisanem prostych rzeczy, ale te proste rzeczy są dość nudne. Teraz chcemy pobrać dane do naszych programów. Jest to trudniejsze, ponieważ musisz nauczyć się dwóch rzeczy, które nie od razu mogą być zrozumiałe, ale zaufaj mi i zrób to tak czy inaczej. Wykonując kolejne ćwiczenia, zrozumiesz, o co chodzi.

Oto czynności, jakie głównie wykonuje oprogramowanie.

1. Przyjmuje pewnego rodzaju dane wejściowe od użytkownika.
2. Zmienia je.
3. Drukuje coś, aby pokazać, co się zmieniło.

Do tej pory drukowałeś łańcuchy znaków, ale nie byłeś w stanie pobrać żadnych danych wejściowych od użytkownika. Być może nawet nie wiesz, czym są „dane wejściowe”, ale wpisz ten kod i przypilnuj, żeby był dokładnie taki sam. W następnym ćwiczeniu zrobimy więcej rzeczy, które pomogą zrozumieć pojęcie danych wejściowych.

ex11.py

```
1 print("Ile masz lat?", end=' ')
2 age = input()
3 print("Ile masz wzrostu?", end=' ')
4 height = input()
5 print("Ile ważysz?", end=' ')
6 weight = input()
7
8 print(f"Więc masz {age} lat, {height} wzrostu i ważysz {weight}.")
```

OSTRZEŻENIE! Na końcu każdej linii print umieszczamy `end=' '`. To informacja dla polecenia print, aby nie kończyć znakiem nowej linii i nie przechodzić do następnej linii.

Co powinieneś zobaczyć

Ćwiczenie 11. — sesja

```
$ python3.6 ex11.py
Ile masz lat? 38
Ile masz wzrostu? 188cm
Ile ważysz? 82kg
Więc masz 38 lat, 188cm wzrostu i ważysz 82kg.
```

Zrób to sam

1. Poszukaj w internecie, co robi funkcja `input` Pythona.
2. Czy możesz znaleźć inne sposoby jej zastosowania? Wypróbuj kilka przykładów, które znajdziesz.
3. Napisz kolejny tego typu „formularz”, aby zadać kilka innych pytań.

Typowe pytania

Jak mogę uzyskać od użytkownika liczbę, abym mógł wykonać operację arytmetyczną? To trochę zaawansowane, ale spróbuj `x = int(input())`, co pobiera z `input()` liczbę jako łańcuch znaków, a następnie konwertuje go na liczbę całkowitą za pomocą `int()`.

Wstawiam mój wzrost do surowych danych wejściowych w ten sposób: `input(188cm)`, ale to nie działa. Nie wstawiaj swojego wzrostu w tym miejscu; wpisz go bezpośrednio w terminalu. Najpierw wróć do kodu i popraw go, żeby był dokładnie taki jak mój. Następnie uruchom skrypt, a gdy się zatrzyma, wpisz swój wzrost na klawiaturze. To wszystko.

Wyświetlanie odpowiedzi dla użytkownika

Kiedy używałeś funkcji `input()`, wpisywałeś znaki (oraz), czyli nawiasy okrągłe. W podobny sposób używałeś nawiasów klamrowych w sformatowanych łańcuchach znaków z dodatkowymi zmiennymi, na przykład `f"{x} {y}"`. W przypadku funkcji `input` możesz również wyświetlić podpowiedź, aby pokazać użytkownikowi, co ma wpisać. Pomiędzy znakami `()` wpisz tekst podpowiedzi, aby wyglądał tak:

```
y = input("Nazwisko? ")
```

Spowoduje to wyświetlenie użytkownikowi podpowiedzi "Nazwisko?" i umieszczenie wyniku w zmiennej `y`. W ten sposób zadajesz użytkownikowi pytanie i uzyskujesz odpowiedź.

Oznacza to, że możemy całkowicie przepisać nasze poprzednie ćwiczenie, używając jedynie funkcji `input` dla wyświetlenia wszystkich podpowiedzi.

ex12.py

```
1 age = input("Ile masz lat? ")
2 height = input("Ile masz wzrostu? ")
3 weight = input("Ile ważysz? ")
4
5 print(f"Więc masz {age} lat, {height} wzrostu i ważysz {weight}.")
```

Co powinieneś zobaczyć

Ćwiczenie 12. — sesja

```
$ python3.6 ex12.py
Ile masz lat? 38
Ile masz wzrostu? 188cm
Ile ważysz? 82kg
Więc masz 38 lat, 188cm wzrostu i ważysz 82kg.
```

Zrób to sam

1. W terminalu, gdzie normalnie wpisujesz polecenie `python3.6`, aby uruchomić swoje skrypty, wpisz polecenie `pydoc input`. Przeczytaj wyświetlone informacje. W systemie Windows użyj polecenia `python -m pydoc input`.
2. Wyjdź z `pydoc`, wpisując `q` (w systemie Windows nie musisz tego robić).

3. Poszukaj w internecie, co robi polecenie `pydoc`.
4. Użyj polecenia `pydoc`, aby przeczytać także o funkcjach `open`, `file`, `os` i module `sys`. Nie przejmuj się, jeśli ich nie rozumiesz; po prostu poczytaj i zanotuj interesujące rzeczy.

Typowe pytania

Dlaczego po wpisaniu polecenia `pydoc` otrzymuję błąd `SyntaxError: invalid syntax`? Nie uruchamiasz `pydoc` z wiersza poleceń; prawdopodobnie uruchamiasz je z poziomu `python3.6`. Najpierw musisz wyjść z Pythona 3.6.

Dlaczego u mnie `pydoc` nie zatrzymuje się, tak jak u Ciebie? Czasami, jeśli dokument pomocy jest odpowiednio krótki, zmieści się na jednym ekranie. Wtedy `pydoc` po prostu go wydrukuje.

Kiedy uruchamiam `pydoc`, otrzymuję komunikat `more is not recognized as an internal`. Niektóre wersje systemu Windows nie mają tego polecenia, co oznacza, że nie będziesz mógł skorzystać z `pydoc`. Możesz pominąć to ćwiczenie z podrozdziału „Zrób to sam” i po prostu poszukać dokumentacji Pythona w internecie, gdy jej potrzebujesz.

Dlaczego nie mogę napisać `print("Ile masz lat?" , input())`? Możesz, ale wtedy wynik wywołania `input()` nie zostanie zapisany w zmiennej i będzie działało w dziwny sposób. Wypróbuj to i następnie spróbuj wydrukować, co wpisujesz. Sprawdź, czy potrafisz zdebugować, dlaczego to nie działa.

Parametry, rozpakowywanie i zmienne

W tym ćwiczeniu omówię jeszcze jedną metodę wprowadzania danych, za pomocą której można przekazywać zmienne do skryptu (*skrypt* to inna nazwa pliku *.py*). Wiesz, że aby uruchomić plik *ex13.py*, musisz wpisać polecenie `python3.6 ex13.py`, prawda? Część *ex13.py* polecenia jest nazywana **argumentem**. Teraz napiszemy skrypt, który przyjmuje również argumenty.

Wpisz ten program, a ja wyjaśnię szczegóły.

ex13.py

```
1  from sys import argv
2  # instrukcje uruchomienia znajdziesz w podrozdziale „Co powinieneś zobaczyć”
3  script, first, second, third = argv
4
5  print("Ten skrypt nazywa się:", script)
6  print("Pierwsza zmienna to:", first)
7  print("Druga zmienna to:", second)
8  print("Trzecia zmienna to:", third)
```

W linii 1. mamy tak zwane **importowanie**. W ten sposób dodajemy do skryptu funkcjonalności z zestawu funkcjonalności Pythona. Zamiast oferować wszystkie funkcjonalności na raz, Python wymaga określenia, czego planujesz użyć. Dzięki temu Twoje programy pozostają niewielkie, ale działa to również jako dokumentacja dla innych programistów, którzy będą czytać Twój kod później.

`argv` jest **zmienną argumentów**. W programowaniu jest to bardzo standardowa nazwa, używana również w wielu innych językach. Zmienna ta przechowuje argumenty przekazywane do skryptu Pythona podczas jego uruchamiania. W ćwiczeniach pobawisz się tym trochę i zobaczysz, co się będzie działo.

Linia 3. **rozpakowuje** `argv`, aby wykonać przypisanie do czterech zmiennych, z którymi możesz pracować: `script` (skrypt), `first` (pierwsza), `second` (druga) i `third` (trzecia), zamiast przechowywać wszystkie argumenty. Może to wyglądać dziwnie, ale „rozpakowuje” jest prawdopodobnie najlepszym słowem opisującym, na czym ta czynność polega. Oznacza to po prostu: „Weź to, co znajduje się w `argv`, rozpakuj i przypisz do wszystkich tych zmiennych po lewej stronie we wskazanej kolejności”.

Potem po prostu wydrukujemy te zmienne, jak zwykle.

Chwileczkę!

Funkcjonalności mają jeszcze inną nazwę

Nazywam je tutaj „funkcjonalnościami” (są to te małe rzeczy, które importujesz, aby Twój program w Pythonie robił więcej), ale nikt inny nie używa takiego określenia. Skorzystałem z takiej nazwy, ponieważ musiałem Cię nakłonić, żebyś nauczył się, czym one są, bez stosowania żargonu programistycznego. Zanim będziesz mógł kontynuować, musisz poznać ich prawdziwą nazwę — są to **moduły**.

Odtąd będziemy nazywać te importowane „funkcjonalności” *modułami*. Będę Cię na przykład instruował: „Powinieneś zaimportować moduł `sys`”. Są one również nazywane przez niektórych programistów „bibliotekami”, ale trzymajmy się po prostu określenia moduły.

Co powinieneś zobaczyć

OSTRZEŻENIE! Zwróć uwagę, że do tej pory uruchamiałeś skrypty Pythona bez argumentów wiersza poleceń. Jeśli wpiszesz tylko `python3.6 ex13.py`, *zrobisz to źle!* Przyjrzyj się uważnie, jak uruchamiam ten skrypt. Odnosi się to do każdego przypadku, gdy używana jest zmienna `argv`.

Uruchom program w ten sposób (*musisz* podać trzy argumenty wiersza poleceń).

Ćwiczenie 13. — sesja

```
$ python3.6 ex13.py pierwsza 2. 3.  
Ten skrypt nazywa się: ex13.py  
Pierwsza zmienna to: pierwsza  
Druga zmienna to: 2.  
Trzecia zmienna to: 3.
```

A to powinieneś zobaczyć, gdy uruchomisz ten skrypt jeszcze kilka razy z różnymi argumentami.

Ćwiczenie 13. — sesja

```
$ python3.6 ex13.py coś rzeczy ta  
Ten skrypt nazywa się: ex13.py  
Pierwsza zmienna to: coś  
Druga zmienna to: rzeczy  
Trzecia zmienna to: ta  
$  
$ python3.6 ex13.py jabłko pomarańcza grejpfrut  
Ten skrypt nazywa się: ex13.py  
Pierwsza zmienna to: jabłko  
Druga zmienna to: pomarańcza  
Trzecia zmienna to: grejpfrut
```

Naprawdę możesz zastąpić pierwsza, 2. i 3. dowolnymi trzema wartościami.

Jeśli nie uruchomisz tego programu poprawnie, otrzymasz następujący błąd.

Ćwiczenie 13. — sesja

```
$ python3.6 ex13.py pierwsza 2.  
Traceback (most recent call last):  
  File "ex13.py", line 3, in <module>  
    script, first, second, third = argv  
ValueError: not enough values to unpack (expected 4, got 3)
```

Dzieje się tak, gdy podczas uruchamiania programu nie wprowadzisz wystarczającej liczby argumentów w poleceniu (w tym przypadku tylko pierwsza 2.). Zwróć uwagę, że kiedy uruchomiłem skrypt z argumentami pierwsza 2., otrzymałem błąd `not enough values to unpack`, który oznacza, że podaliśmy za mało parametrów.

Zrób to sam

1. Spróbuj podczas uruchamiania skryptu podać mniej niż trzy argumenty. Otrzymałeś błąd? Sprawdź, czy potrafisz to wyjaśnić.
2. Napisz jeden skrypt, który ma mniej argumentów, i jeden, który ma więcej. Upewnij się, że nadajesz odpowiednie nazwy rozpakowywanym zmiennym.
3. Połącz `input` z `argv`, aby utworzyć skrypt, który pobiera więcej danych wejściowych od użytkownika. Uważaj, by nie przekombinować. Po prostu pobierz od użytkownika jedną rzecz za pomocą `argv`, a drugą przy użyciu `input`.
4. Pamiętaj, że moduły oferują funkcjonalności. Moduły. Moduły. Zapamiętaj to, ponieważ będziemy potrzebować tego później.

Typowe pytania

Po uruchomieniu otrzymuję błąd `ValueError: not enough values to unpack`. Pamiętaj, że ważną umiejętnością jest zwracanie uwagi na szczegóły. Jeśli zajrzysz do podrozdziału „Co powinieneś zobaczyć”, przekonasz się, że uruchamiam skrypt z parametrami w wierszu poleceń. Powinieneś uruchomić go w dokładnie taki sam sposób.

Jaka jest różnica między `argv` i `input()`? Różnica polega na tym, w którym miejscu użytkownik musi podać dane wejściowe. Jeśli dane wejściowe skryptu mają być wpisywane w wierszu poleceń, używasz `argv`. Jeśli chcesz, aby były one wprowadzane za pomocą klawiatury podczas działania skryptu, używasz `input()`.

Czy argumenty wiersza poleceń są łańcuchami znaków? Tak, są traktowane jak łańcuchy znaków nawet wtedy, jeśli w wierszu poleceń wpisałeś liczby. Aby je przekonwertować, użyj `int()`, na przykład `int(input())`.

Jak korzystać z wiersza poleceń? Do tej pory powinienś już używać go bardzo szybko i płynnie, ale jeśli musisz się tego nauczyć na tym etapie, przeczytaj dodatek „Przyspieszony kurs wiersza poleceń”.

Nie mogę połączyć argv z input(). Nie przekombinuj. Po prostu na końcu tego skryptu wstaw dwie linie używające `input()`, aby coś pobrać, a następnie wydrukować. Kiedy to Ci się uda, możesz pobawić się, używając obu opcji w tym samym skrypcie na więcej sposobów.

Dlaczego nie mogę zrobić tak: `input('? ') = x`? Ponieważ to jest odwrotne do tego, w jaki sposób powinno działać. Zrób to tak, jak robię ja, a wszystko zadziała prawidłowo.

Znak zachęty i przekazywanie zmiennej

Zróbmy jedno ćwiczenie, w którym użyjemy jednocześnie `argv` i `input`, aby zapytać użytkownika o coś konkretnego. Będziemy potrzebować tego do następnego ćwiczenia, w którym nauczysz się odczytywać i zapisywać pliki. W tym ćwiczeniu użyjemy funkcji `input` w nieco inny sposób — będzie wyświetlała prosty znak zachęty `>`. Przypomina to gry, takie jak *Zork* lub *Adventure*.

ex14.py

```
1  from sys import argv
2
3  script, user_name = argv
4  prompt = '> '
5
6  print(f"Cześć, {user_name}, jestem skryptem {script}.")
7  print("Chciałbym Ci zadać kilka pytań.")
8  print(f"Lubisz mnie, {user_name}?")
9  likes = input(prompt)
10
11 print(f"Gdzie mieszkasz, {user_name}?")
12 lives = input(prompt)
13
14 print("Jaki masz komputer?")
15 computer = input(prompt)
16
17 print(f"""
18 Ok, gdy zapytałem, czy mnie lubisz, odpowiedziałeś {likes}.
19 Mieszkasz w {lives}. Nie jestem pewien, gdzie to jest.
20 I masz komputer {computer}. Fajnie.
21 """)
```

Tworzymy zmienną `prompt`, której przypisujemy żądany znak zachęty i przekazujemy ją do funkcji `input`, zamiast wpisywać ten znak zachęty za każdym razem. Gdy teraz będziemy chcieli zmienić znak zachęty na inną odpowiedź, po prostu wykonamy zmianę w tym jednym miejscu i uruchomimy ponownie skrypt. Jest to bardzo przydatne.

Co powinieneś zobaczyć

Pamiętaj, że podczas uruchamiania skryptu musisz podać swoje imię dla argumentów argv.

Ćwiczenie 14. — sesja

```
$ python3.6 ex14.py zed
Cześć, zed, jestem skryptem ex14.py.
Chciałbym Ci zadać kilka pytań.
Lubisz mnie, zed?
> Tak
Gdzie mieszkasz, zed?
> San Francisco
Jaki masz komputer?
> Tandy 1000
```

Ok, gdy zapytałem, czy mnie lubisz, odpowiedziałeś Tak.
Mieszkasz w San Francisco. Nie jestem pewien, gdzie to jest.
I masz komputer Tandy 1000. Fajnie.

Zrób to sam

1. Wspomniałem o grach *Zork* i *Adventure*. Dowiedz się, jakie to były gry. Spróbuj znaleźć ich kopie i zagraj w nie.
2. Zmień wartość zmiennej prompt na coś zupełnie innego.
3. Dodaj kolejny argument i użyj go w swoim skrypcie w taki sam sposób jak w poprzednim ćwiczeniu z `first`, `second` = ARGV.
4. Upewnij się, że rozumiesz, w jaki sposób połączyłem styl `"""` wieloliniowego łańcucha znaków z aktywatorem formatowania `{}` w ostatnim poleceniu `print`.

Typowe pytania

Podczas uruchamiania tego skryptu otrzymuję błąd `SyntaxError: invalid syntax`.

Przypomnę jeszcze raz, że musisz go uruchomić w wierszu poleceń, a nie w Pythonie. Jeśli wpiszesz `python3.6`, a następnie spróbujesz wpisać `python3.6 ex14.py zed`, otrzymasz błąd, ponieważ uruchamiasz Pythona *wewnątrz* Pythona. Zamknij okno wiersza poleceń, uruchom ponownie, a następnie po prostu wpisz `python3.6 ex14.py zed`.

Nie rozumiem, co masz na myśli, gdy mówisz o zmianie znaku zachęty? Zobacz zmienną `prompt` = `'> '`. Nadaj jej inną wartość. Znasz już to; to tylko łańcuch znaków i wykonałeś 13 ćwiczeń, tworząc takie łańcuchy, więc poświęć chwilę, aby to rozgryźć.

Otrzymuję błąd `ValueError: not enough values to unpack`. Pamiętasz, jak kazałem Ci zajrzeć do podrozdziału „Co powinienes zobaczyć” i powtórzyć to, co tam zrobiłem? Tutaj musisz zrobić to samo i skupić się na tym, w jaki sposób wpisuję polecenie i dlaczego mam argument wiersza poleceń.

Jak mogę uruchomić to za pomocą IDLE? Nie używaj IDLE.

Czy mogę użyć podwójnych cudzysłówów dla zmiennej `prompt`? Oczywiście. Śmiało, spróbuj tak zrobić.

Masz komputer Tandy? Miałem, gdy byłem dzieckiem.

Podczas uruchamiania programu otrzymuję błąd `NameError: name 'prompt' is not defined`. Prawdopodobnie zrobiłeś literówkę w nazwie zmiennej `prompt` lub zapomniałeś wpisać tę linię. Wróć do skryptu i porównaj każdą linię swojego kodu z moim, zaczynając od dołu skryptu i idąc w górę. Za każdym razem, gdy otrzymujesz taki błąd, oznacza to, że coś źle napisałeś lub zapomniałeś utworzyć zmienną.

Czytanie z plików

Wiesz już, jak uzyskać dane wejściowe od użytkownika za pomocą `input` lub `argv`. Teraz nauczysz się, jak czytać z pliku. Być może będziesz musiał spędzić nad tym ćwiczeniem trochę więcej czasu niż zwykle, aby zrozumieć, co się w nim dzieje, więc wykonaj je uważnie i pamiętaj o sprawdzeniu kodu. Praca z plikami to łatwy sposób na *wykasowanie własnej pracy*, jeśli nie jesteś ostrożny.

Ćwiczenie to wymaga napisania dwóch plików. Jeden to standardowy plik `ex15.py`, który będziesz uruchamiał, natomiast drugi nazywa się `ex15_sample.txt`. Ten drugi plik nie jest skryptem, ale zwykłym plikiem tekstowym, który będziemy odczytywać w naszym skrypcie. Oto zawartość tego pliku.

```
To są rzeczy, które wpisałem w pliku.  
To naprawdę fajne rzeczy.  
Dostarczą Ci dużo radości i zabawy.
```

Chcemy otworzyć ten plik w naszym skrypcie i wydrukować go. Nie chcemy jednak po prostu zakodować na sztywno nazwy `ex15_sample.txt` w skrypcie. „Kodowanie na sztywno” oznacza umieszczenie bezpośrednio w kodzie źródłowym łańcucha znaków reprezentującego pewne informacje, które powinny pochodzić od użytkownika. Takie podejście jest złe, ponieważ chcemy, aby skrypt później wczytał inne pliki. Rozwiązaniem jest użycie `argv` lub `input`, aby zapytać użytkownika, który plik otworzyć, zamiast na sztywno kodować nazwę pliku.

ex15.py

```
1  from sys import argv  
2  
3  script, filename = argv  
4  
5  txt = open(filename)  
6  
7  print(f'Oto Twój plik {filename}:')  
8  print(txt.read())  
9  
10 print("Wpisz ponownie nazwę pliku:")  
11 file_again = input("> ")  
12  
13 txt_again = open(file_again)  
14  
15 print(txt_again.read())
```

W tym pliku dzieje się kilka ciekawych rzeczy, więc przeanalizujmy je szybko.

- Linie od 1. do 3. wykorzystują `argv` do pobrania nazwy pliku. Następnie w linii 5. używamy nowego polecenia, czyli `open`. Uruchom teraz polecenie `pydoc open` i przeczytaj instrukcję. Zwróć uwagę, że podobnie jak Twoje skrypty oraz `input`, to polecenia pobiera parametr i zwraca wartość, którą możesz przypisać do własnej zmiennej. Właśnie otworzyłeś plik.

- Linia 7. drukuje krótki komunikat, ale w linii 8. mamy coś nowego i bardzo ekscytującego. Na zmiennej `txt` wywołujemy funkcję o nazwie `read`. Z polecenia `open` otrzymujemy plik, na którym także możemy wykonywać pewne polecenia. Robimy to poprzez dodanie do pliku kropki (`.`), nazwy polecenia i parametrów, podobnie jak w przypadku `open` i `input`. Różnica polega na tym, że `txt.read()` mówi: „Hej, `txt`! Wykonaj polecenie `read` bez parametrów!”.

Pozostała część pliku jest podobna, ale analizę wykonasz samodzielnie w podrozdziale „Zrób to sam”.

Co powinieneś zobaczyć

OSTRZEŻENIE! Skup się! Powtarzam, *skup się!* Wcześniej uruchamiałeś programy jedynie za pomocą nazwy skryptu, ale teraz, gdy używasz `argv`, musisz dodać argumenty. Spójrz na pierwszą linię poniższego przykładu, a zobaczysz, że wpisuję `python ex15.py ex15_sample.txt`, aby go uruchomić. Zwróć uwagę na dodatkowy argument `ex15_sample.txt` po nazwie skryptu `ex15.py`. Jeśli tego nie wpiszesz, otrzymasz błąd, więc skup się!

Utworzyłem plik o nazwie `ex15_sample.txt` i uruchomiłem skrypt.

Ćwiczenie 15. — sesja

```
$ python3.6 ex15.py ex15_sample.txt
Oto Twój plik ex15_sample.txt:
To są rzeczy, które wpisałem w pliku.
To naprawdę fajne rzeczy.
Dostarczą Ci dużo radości i zabawy.
```

```
Wpisz ponownie nazwę pliku:
> ex15_sample.txt
To są rzeczy, które wpisałem w pliku.
To naprawdę fajne rzeczy.
Dostarczą Ci dużo radości i zabawy.
```

Zrób to sam

To jest duży skok, więc zanim przejdiesz dalej, upewnij się, że wykonałeś ćwiczenie samodzielnie, najlepiej jak potrafisz.

1. Nad każdą linią napisz komentarz opisujący, co ona robi.
2. Jeśli nie masz pewności, poproś kogoś o pomoc lub poszukaj informacji w internecie. W wielu przypadkach wpisanie w wyszukiwarce „python3.6 COŚTAM” pozwoli Ci znaleźć odpowiedź na pytanie, co COŚTAM robi w Pythonie. Spróbuj wyszukać „python3.6 open”.

3. Użyłem tutaj słowa „polecenia”, ale polecenia są również nazywane „funkcjami” i „metodami”. O funkcjach i metodach będziesz uczył się dalej w tej książce.
4. Pozbądź się linii od 10. do 15., w których używasz `input`, i ponownie uruchom skrypt.
5. Spróbuj napisać ten skrypt, używając tylko funkcji `input`. Dlaczego jeden sposób uzyskania nazwy pliku jest lepszy od drugiego?
6. Uruchom polecenie `python3.6`, aby wywołać powłokę Pythona, i użyj polecenia `open` z wiersza poleceń w taki sposób, jak robiłeś to w programie. Zwróciłeś uwagę, że możesz otwierać pliki i uruchamiać na nich polecenie `read` z poziomu `python3.6`?
7. Niech Twój skrypt wywoła również funkcję `close()` na zmiennych `txt` i `txt_again`. Ważne jest, aby zamykać pliki po zakończeniu pracy z nimi.

Typowe pytania

Czy `txt = open(filename)` zwraca zawartość pliku? Nie, nie zwraca. W rzeczywistości tworzy tak zwany *obiekt pliku*. Możesz potraktować plik jak stary napęd taśmowy używany w komputerach mainframe w latach 50. ubiegłego wieku lub nawet jak odtwarzacz DVD wykorzystywany obecnie. Możesz poruszać się wewnątrz nich, a następnie „odczytywać” je, ale odtwarzacz DVD nie jest dyskiem DVD w taki sam sposób, jak obiekt pliku nie jest zawartością pliku.

Nie mogę wpisywać kodu w terminalu (programie PowerShell), tak jak napisano w ćwiczeniu 6. w podrozdziale „Zrób to sam”. Najpierw w wierszu poleceń wpisz po prostu `python3.6` i naciśnij *Enter*. Znajdziesz się w powłoce `python3.6`, tak jak robiliśmy już kilka razy wcześniej. Teraz możesz wpisywać kod, a Python uruchomi go w małych kawałkach. Pobaw się tym. Aby wyjść z powłoki, wpisz `quit()` i wciśnij *Enter*.

Dlaczego nie otrzymujemy błędu, gdy dwukrotnie otwieramy plik? Python nie będzie ograniczał możliwości otwierania pliku więcej niż raz, a czasami jest to konieczne.

Co znaczy `from sys import argv`? Na razie wystarczy, jeśli będziesz wiedział, że `sys` to pakiet, a ta fraza mówi po prostu, aby pobrać funkcjonalność `argv` z tego pakietu. Więcej na ten temat dowiesz się później.

Wpisuję nazwę pliku jako `script`, `ex15_sample.txt` = `argv`, ale to nie działa. Nie, w ten sposób się tego nie robi. Wpisz dokładnie taki sam kod jak mój, a następnie uruchom go z wiersza poleceń dokładnie tak samo jak ja. Nie umieszczaj w nim nazw plików — pozwól, aby Python wprowadził nazwę.

Czytanie i zapisywanie plików

Jeśli wykonałeś zadania z podrozdziału „Zrób to sam” z ostatniego ćwiczenia, poznałeś różnego rodzaju polecenia (metody/funkcje), których możesz używać na plikach. Oto lista poleceń, które powinieneś zapamiętać.

- `close`: zamyka plik, tak jak polecenie *File/Save* (plik/zapisz) w edytorze.
- `read`: czyta zawartość pliku; wynik możesz przypisać do zmiennej.
- `readline`: czyta tylko jedną linię pliku tekstowego.
- `truncate`: opróżnia plik; bądź ostrożny z tym poleceniem, jeśli pracujesz z ważnym plikiem.
- `write('jakiś_tekst')`: zapisuje *'jakiś_tekst'* w pliku.
- `seek(0)`: przesuwa lokalizację odczytu/zapisu na początek pliku.

Jednym ze sposobów zapamiętania tego, co robi każde z tych poleceń, jest użycie analogii do płyty winylowej, kasety magnetofonowej, kasety VHS albo odtwarzacza DVD lub CD. W początkach istnienia komputerów dane były przechowywane na każdym z tych rodzajów mediów, dlatego wiele operacji na plikach wciąż przypomina liniowy system pamięci masowej. Napędy taśmowe i DVD muszą „poszukać” określonego miejsca, a następnie mogą odczytywać lub zapisywać w tym miejscu. Dzisiaj mamy systemy operacyjne i nośniki pamięci, które zacierają granice między pamięcią o swobodnym dostępie a napędami dyskowymi, ale nadal używamy starszej koncepcji liniowej taśmy z ruchomą głowicą odczytu/zapisu.

Na razie są to istotne polecenia, które musisz znać. Niektóre z nich przyjmują parametry, ale nie przejmuj się tym. Musisz tylko zapamiętać, że `write` przyjmuje parametr w postaci łańcucha znaków, który chcesz zapisać w pliku.

Wykorzystajmy niektóre z tych poleceń, aby utworzyć prosty edytor tekstu.

ex16.py

```
1  from sys import argv
2
3  script, filename = argv
4
5  print(f"Wymażemy {filename}.")
6  print("Jeśli tego nie chcesz, wciśnij CTRL+C (^C).")
7  print("Jeśli tego chcesz, wciśnij RETURN.")
8
9  input("?")
10
11 print("Otwieranie pliku...")
12 target = open(filename, 'w')
13
14 print("Wykasowywanie pliku. Do widzenia!")
15 target.truncate()
```

```
16
17     print("Teraz poproszę Cię o wpisanie trzech linii tekstu.")
18
19     line1 = input("linia 1: ")
20     line2 = input("linia 2: ")
21     line3 = input("linia 3: ")
22
23     print("Zapiszę je w pliku.")
24
25     target.write(line1)
26     target.write("\n")
27     target.write(line2)
28     target.write("\n")
29     target.write(line3)
30     target.write("\n")
31
32     print("A na koniec zamykamy plik.")
33     target.close()
```

To duży plik, prawdopodobnie największy, jaki wpisałeś. Dlatego nie spiesz się, sprawdź kod i uruchom plik. Jedną ze sztuczek jest uruchamianie po kolei fragmentów kodu. Uruchom linie od 1. do 8., następnie kolejne pięć linii, potem jeszcze kilka, aż otrzymasz działającą całość.

Co powinieneś zobaczyć

Zobaczysz dwie rzeczy. Najpierw będzie to wynik wykonania Twojego nowego skryptu.

Ćwiczenie 16. — sesja

```
$ python3.6 ex16.py test.txt
Wymażemy test.txt.
Jeśli tego nie chcesz, wciśnij CTRL+C (^C).
Jeśli tego chcesz, wciśnij RETURN.
?
Otwieranie pliku...
Wykasowywanie pliku. Do widzenia!
Teraz poproszę Cię o wpisanie trzech linii tekstu.
linia 1: Mary małą owcę ma
linia 2: Która biała jest jak mgła
linia 3: I była też bardzo smaczna
Zapiszę je w pliku.
A na koniec zamykamy plik.
```

Teraz otwórz w edytorze plik, który utworzyłeś (w moim przypadku `test.txt`) i sprawdź go. Zgrabne, prawda?

Zrób to sam

1. Jeśli tego nie rozumiesz, wróć do początku kodu i użyj sztuczki z komentarzami, aby poukładać to sobie w głowie. Jeden prosty komentarz nad każdą linią pomoże Ci zrozumieć kod lub przynajmniej pokaże, jakich informacji powinieneś poszukać.
2. Napisz skrypt podobny do ostatniego ćwiczenia, wykorzystujący `read` i `argv` do odczytania pliku, który właśnie utworzyłeś.
3. W naszym pliku jest zbyt wiele powtórzeń. Użyj łańcuchów znaków, formatowania i sekwencji ucieczki, aby wydrukować `line1`, `line2` i `line3` za pomocą tylko jednego polecenia `target.write()` zamiast sześciu.
4. Dowiedz się, dlaczego musieliśmy do polecenia `open` przekazać `'w'` jako dodatkowy parametr. Podpowiedź: polecenie `open` ma zabezpieczenie, które wymusza wyraźne określenie, że chcesz zapisać plik.
5. Jeśli otworzysz plik w trybie `'w'`, to czy naprawdę potrzebujesz `target.truncate()`? Przeczytaj dokumentację Pythona dla funkcji `open` i przekonaj się, czy to prawda.

Typowe pytania

Czy `truncate()` jest konieczne przy parametrze `'w'`? Zobacz ćwiczenie 5. w podrzdziale „Zrób to sam”.

Co oznacza `'w'`? To tylko łańcuch znaków, który określa rodzaj trybu dla pliku. Jeśli używasz `'w'`, wydajesz polecenie „otwórz ten plik w trybie zapisu (ang. *write*)”, stąd litera „w”. Istnieją również tryby odczytu (ang. *read*) — `'r'`, dołączania (ang. *append*) — `'a'` oraz ich modyfikacje.

Jakich modyfikatorów trybu pliku mogę używać? Najważniejszym modyfikatorem, który powinieneś teraz poznać, jest modyfikator `+`, więc możesz napisać `'w +'`, `'r +'` i `'a +'`. Spowoduje to otwarcie pliku w trybie odczytu i zapisu, i w zależności od użytego znaku odpowiednio wypozycjonuje plik.

Czy wpisanie po prostu `open(filename)` otwiera plik w trybie `'r'` (odczytu)? Tak, jest to domyślne zachowanie dla funkcji `open()`.

Więcej plików

Pobawmy się jeszcze trochę plikami. Napiszmy skrypt Pythona kopiujący jeden plik do drugiego. Będzie on bardzo krótki, ale da Ci wyobrażenie na temat innych rzeczy, które możesz zrobić z plikami.

ex17.py

```
1  from sys import argv
2  from os.path import exists
3
4  script, from_file, to_file = argv
5
6  print(f"Kopiowanie z {from_file} do {to_file}")
7
8  # moglibyśmy zrobić te dwie rzeczy w jednej linii? jak?
9  in_file = open(from_file)
10 indata = in_file.read()
11
12 print(f"Plik wejściowy ma {len(indata)} bajtów")
13
14 print(f"Czy plik wyjściowy istnieje? {exists(to_file)}")
15 print("Wciśnij RETURN, aby kontynuować lub CTRL+C, żeby anulować.")
16 input()
17
18 out_file = open(to_file, 'w')
19 out_file.write(indata)
20
21 print("W porządku, zrobione.")
22
23 out_file.close()
24 in_file.close()
```

Powinieneś natychmiast zauważyć, że importujemy kolejne przydatne polecenie o nazwie `exists`. Jeśli plik istnieje, zwraca ono `True` na podstawie nazwy pliku podanej jako argument w łańcuchu znaków. Jeżeli plik nie istnieje, zwraca `False`. Będziemy używać tej funkcji w drugiej połowie książki do wielu rzeczy, ale teraz powinieneś zobaczyć, jak można ją zaimportować.

Używanie polecenia `import` jest sposobem na uzyskanie dużej ilości wolnego kodu, który został napisany przez innych, lepszych (no, zazwyczaj) programistów, więc nie trzeba go już pisać samemu.

Co powinieneś zobaczyć

Podobnie jak inne skrypty, uruchom ten program z dwoma argumentami: źródłowym plikiem do skopiowania i plikiem docelowym. Użyj prostego pliku testowego o nazwie `test.txt`:

Ćwiczenie 17. — sesja

```
$ # najpierw utwórzmy prosty plik
$ echo "To jest plik testowy." > test.txt
$ # następnie podejrzymy zawartość pliku
$ cat test.txt
To jest plik testowy.
$ # teraz uruchommy na nim nasz skrypt
$ python3.6 ex17.py test.txt new_file.txt
Kopiowanie z test.txt do new_file.txt
Plik wejściowy ma 48 bajtów
Czy plik wyjściowy istnieje? False
Wciśnij RETURN, aby kontynuować lub CTRL+C, żeby anulować.
```

W porządku, zrobione.

Powinno to działać z dowolnym plikiem. Spróbuj jeszcze kilka razy i zobacz, co się będzie działo. Tylko uważaj, żeby nie skasować jakiegoś ważnego pliku.

OSTRZEŻENIE! Widziałeś sztuczkę, którą zrobiłem z poleceniem echo, aby utworzyć plik, i poleceniem cat, żeby wyświetlić plik? Możesz nauczyć się tego, korzystając z dodatku „Przyspieszony kurs wiersza poleceń”.

Zrób to sam

1. Ten skrypt jest *naprawdę* denerwujący. Nie musisz pytać użytkownika przed zrobieniem kopii, a ponadto na ekranie drukowanych jest zbyt dużo rzeczy. Spraw, by ten skrypt stał się bardziej przyjazny w użyciu, usuwając pewne funkcjonalności.
2. Sprawdź, jak krótki skrypt możesz napisać. Ja mógłbym zrobić to w jednej linii.
3. Zwróciłeś uwagę, że w podrozdziale „Co powinieneś zobaczyć” użyłem czegoś, co jest nazywane cat? Jest to stare polecenie, które konkatenuje pliki, ale przede wszystkim jest to łatwy sposób na wydrukowanie pliku na ekranie. Wpisz man cat, aby o nim poczytać.
4. Dowiedz się, dlaczego musiałeś napisać w kodzie out_file.close().
5. Poczytaj o instrukcji import Pythona i uruchom python3.6, aby ją wypróbować. Spróbuj zaimportować różne rzeczy i zobacz, czy robisz to prawidłowo. Nie przejmuj się, jeśli Ci się nie uda.

Typowe pytania

Dlaczego 'w' jest w cudzysłowie? To łańcuch znaków. Używasz łańcuchów znaków już od pewnego czasu. Upewnij się, że wiesz, co to jest łańcuch znaków.

Nie ma mowy, żebyś mógł zrobić to w jednej linii! To; zależy; od; tego; jak; zdefiniujemy; jedną; linię; kodu.

Czy to normalne, że to ćwiczenie wydaje mi się naprawdę trudne? Tak, to całkowicie normalne. Programowanie może u Ciebie nie „zaskoczyć” chociażby nawet aż do ćwiczenia 36. albo dopóki nie skończysz książki, a potem nie utworzysz czegoś za pomocą Pythona. Każdy jest inny, więc idź dalej i wracaj do ćwiczeń, z którymi miałeś problemy, dopóki nie zrozumiesz. Bądź cierpliwy.

Do czego służy funkcja `len()`? Pobiera długość przekazanego do niej łańcucha znaków, a następnie zwraca ją jako liczbę. Pobaw się tą funkcją.

Kiedy próbuję skrócić ten skrypt, pojawia się błąd przy zamykaniu plików na końcu. Prawdopodobnie zrobiłeś coś takiego jak `indata = open(from_file).read()`, co oznacza, że nie musisz wtedy robić `in_file.close()`, kiedy dojdiesz do końca skryptu. Po uruchomieniu tej linii plik powinien już zostać zamknięty przez Pythona.

Otrzymuję błąd `SyntaxError: EOL while scanning string literal`. Zapomniałeś odpowiednio zakończyć łańcuch znaków cudzysłowem. Wróć do tej linii i sprawdź ją.

Nazwy, zmienne, kod i funkcje

Poważny tytuł, prawda? Zamierzam wprowadzić **funkcje**. Tadam! Każdy programista może w nieskończoność mówić o funkcjach i różnych pomysłach na temat tego, w jaki sposób działają i co robią, ale przedstawię Ci najprostsze wyjaśnienie, którego możesz teraz użyć.

Funkcje robią trzy rzeczy.

1. Nazywają fragmenty kodu w taki sposób, w jaki zmienne nazywają łańcuchy znaków i liczby.
2. Przyjmują argumenty w taki sposób, w jaki Twoje skrypty przyjmują argv.
3. Dzięki powyższym dwóm funkcjonalnościom pozwalają Ci tworzyć własne „miskrypty” lub „niewielkie polecenia”.

W Pythonie funkcję możesz utworzyć za pomocą słowa `def`. Napiszesz cztery różne funkcje, które działają jak skrypty, a następnie pokażę Ci, jak są one powiązane.

ex18.py

```

1  # ta funkcja jest podobna do skryptów z argv
2  def print_two(*args):
3      arg1, arg2 = args
4      print(f"arg1: {arg1}, arg2: {arg2}")
5
6  # ok, to *args jest właściwie bezcelowe; możemy po prostu zrobić tak
7  def print_two_again(arg1, arg2):
8      print(f"arg1: {arg1}, arg2: {arg2}")
9
10 # ta funkcja przyjmuje po prostu jeden argument
11 def print_one(arg1):
12     print(f"arg1: {arg1}")
13
14 # ta funkcja nie przyjmuje żadnych argumentów
15 def print_none():
16     print("Ja nie mam nic.")
17
18
19 print_two("Zed", "Shaw")
20 print_two_again("Zed", "Shaw")
21 print_one("Pierwszy!")
22 print_none()
```

Przeanalizujmy pierwszą funkcję, `print_two`, która jest najbardziej podobna do tego, co już znasz, a co dotyczyło skryptów.

1. Za pomocą słowa `def` (jak definicja) mówimy Pythonowi, że chcemy utworzyć funkcję.

2. W tej samej linii, co `def`, podajemy nazwę funkcji. W tym przypadku nazwaliśmy ją po prostu `print_two` (drukuj dwa), ale mogłaby to być dowolna inna nazwa. Nie ma to znaczenia, z tym wyjątkiem, że funkcja powinna mieć krótką nazwę, określającą, co robi.
3. Instruuemy funkcję, że chcemy `*args` (gwiazdka, args), co jest podobne do parametru `argv`, ale dla funkcji. Żeby wszystko zadziało poprawnie, **musimy** umieścić to w nawiasach.
4. Kończymy tę linię dwukropkiem (`:`), a następną zaczynamy od wcięcia.
5. Po dwukropku wszystkie linie, które mają cztery spacje wcięcia, zostaną dołączone do `print_two`. Nasza pierwsza linia z wcięciem rozpakowuje argumenty, tak samo jak w przypadku skryptów.
6. Aby pokazać, jak to działa, drukujemy argumenty, tak jak w skryptach.

Problem z `print_two` polega na tym, że nie jest to najprostszy sposób na utworzenie funkcji. W Pythonie możemy pominąć rozpakowywanie argumentów i po prostu użyć pożądanых nazw bezpośrednio w nawiasach. To właśnie robi `print_two_again`.

Po tym masz przykład `print_one`, który pokazuje, jak utworzyć funkcję przyjmującą jeden argument.

Na koniec masz funkcję `print_none`, która nie ma żadnych argumentów.

OSTRZEŻENIE! To bardzo ważne. *Nie zniechęcaj się od razu, jeśli nie wszystko ma dla Ciebie sens. Wykonamy kilka ćwiczeń łączących funkcje ze skryptami i pokażę Ci, jak zrobić więcej. Na razie po prostu myśl „miniskrypt”, kiedy mówię „funkcja”, i pobaw się funkcjami.*

Co powinieneś zobaczyć

Jeśli uruchomisz plik `ex18.py`, powinieneś zobaczyć następujący listing.

Ćwiczenie 18. — sesja

```
$ python3.6 ex18.py
arg1: Zed, arg2: Shaw
arg1: Zed, arg2: Shaw
arg1: Pierwszy!
Ja nie mam nic.
```

Od razu widać, jak działa funkcja. Zwróć uwagę, że zastosowałeś funkcję w taki sposób, w jaki używasz `exists`, `open` i innych poleceń. Właściwie to Cię oszukałem, ponieważ w Pythonie te polecenia są po prostu funkcjami. Oznacza to, że możesz tworzyć własne polecenia i również używać ich w skryptach.

Zrób to sam

Utwórz *listę kontrolną funkcji* do późniejszych ćwiczeń. Spisz poniższe punkty na fiszce i używaj jej jako ściągawki podczas wykonywania pozostałych ćwiczeń lub do momentu, gdy nie będziesz już jej potrzebował.

1. Czy rozpocząłeś definicję funkcji od słowa `def`?
2. Czy Twoja nazwa funkcji zawiera tylko znaki i podkreślniki (`_`)?
3. Czy wstawiłeś nawias otwierający bezpośrednio po nazwie funkcji?
4. Czy umieściłeś argumenty po nawiasie, oddzielając je przecinkami?
5. Czy każdy argument jest unikatowy (czyli nie ma zduplikowanych nazw)?
6. Czy bezpośrednio po argumentach wstawiłeś nawias zamykający i dwukropek?
7. Czy wszystkie linie kodu, które mają należeć do funkcji, wciąłeś o cztery spacje (nie więcej i nie mniej)?
8. Czy „zakończyłeś” funkcję, powracając do pisania bez wcięć?

Podczas uruchamiania (inaczej używania lub wywoływania) funkcji sprawdź następujące rzeczy.

1. Czy wywołałeś (zastosowałeś, uruchomiłeś) tę funkcję, wpisując jej nazwę?
2. Czy umieściłeś znak `(` po nazwie funkcji, żeby ją uruchomić?
3. Czy umieściłeś żądane wartości w nawiasach, oddzielając je przecinkami?
4. Czy zakończyłeś wywołanie funkcji znakiem `)`?

Korzystaj z tych dwóch list kontrolnych podczas pozostałych lekcji, aż przyjdzie taki czas, że nie będziesz ich już potrzebować.

Na koniec powtórz sobie kilka razy: funkcję można uruchomić, wywołać lub jej użyć — wszystkie te określenia oznaczają to samo.

Typowe pytania

Co jest dozwolone dla nazw funkcji? To samo, co dla nazw zmiennych. Będzie działać wszystko, co nie zaczyna się od liczby i zawiera litery, liczby oraz znaki podkreślenia.

Jaką rolę pełni gwiazdka w `*args`? Jest to instrukcja dla Pythona, aby wziąć wszystkie argumenty funkcji, a następnie umieścić je w `args` jako listę. Przypomina używany już przez Ciebie `argv`, ale ma zastosowanie do funkcji. Zwykle nie jest to używane zbyt często, chyba że jest szczególnie potrzebne.

To naprawdę nudne i monotonne. To dobrze. Oznacza to, że zaczynasz coraz lepiej radzić sobie z wpisywaniem kodu i rozumieniem, co robi. Aby było mniej nudno, spróbuj celowo popsuć wszystko, co każę Ci wpisywać.

Funkcje i zmienne

Wprowadzenie funkcji mogło stanowić dla Ciebie oszałamiającą ilość informacji, ale nie przejmuj się. Po prostu kontynuuj wykonywanie tych ćwiczeń i przeglądaj listę kontrolną z ostatniego ćwiczenia, a w końcu to załapiesz.

Istnieje pewna drobna kwestia, z której mogłeś nie zdawać sobie sprawy, więc zajmiemy się tym teraz. Zmienne w funkcji nie są połączone ze zmiennymi w skrypcie. Oto ćwiczenie, które przybliży Ci to zagadnienie.

ex19.py

```
1 def cheese_and_crackers(cheese_count, boxes_of_crackers):
2     print(f'Masz {cheese_count} kawałków sera!')
3     print(f'Masz {boxes_of_crackers} paczek krakersów!')
4     print("Stary, to wystarczy, żeby zrobić imprezę!")
5     print("Zorganizuj sobie koc.\n")
6
7
8     print("Możemy po prostu bezpośrednio podać funkcji liczby:")
9     cheese_and_crackers(20, 30)
10
11
12     print("Albo możemy użyć zmiennych ze skryptu:")
13     amount_of_cheese = 10
14     amount_of_crackers = 50
15
16     cheese_and_crackers(amount_of_cheese, amount_of_crackers)
17
18
19     print("Możemy również wykonać wewnątrz operacje arytmetyczne:")
20     cheese_and_crackers(10 + 20, 5 + 6)
21
22
23     print("I możemy połączyć te dwie rzeczy, czyli zmienne i operacje arytmetyczne:")
24     cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

To pokazuje różne możliwości dostarczania naszej funkcji `cheese_and_crackers` (ser i krakersy) żądanych wartości do wydrukowania. Możemy podawać po prostu liczby. Możemy podawać jej zmienne. Możemy zadawać operacje arytmetyczne. Możemy nawet łączyć operacje arytmetyczne i zmienne.

W pewnym sensie argumenty funkcji są podobne do naszego znaku równości (=), gdy tworzymy zmienną. Jeśli możesz użyć znaku równości do nazwania czegoś, zwykle możesz przekazać to do funkcji jako argument.

Co powinieneś zobaczyć

Przestuduj dane wyjściowe z tego skryptu i porównaj je z tym, co — według Ciebie — powinieneś otrzymać dla każdego z przykładów z tego skryptu.

Ćwiczenie 19. — sesja

```
$ python3.6 ex19.py
```

Możemy po prostu bezpośrednio podać funkcji liczby:

Masz 20 kawałków sera!

Masz 30 paczek krakersów!

Stary, to wystarczy, żeby zrobić imprezę!

Zorganizuj sobie koc.

Albo możemy użyć zmiennych ze skryptu:

Masz 10 kawałków sera!

Masz 50 paczek krakersów!

Stary, to wystarczy, żeby zrobić imprezę!

Zorganizuj sobie koc.

Możemy również wykonać wewnątrz operacje arytmetyczne:

Masz 30 kawałków sera!

Masz 11 paczek krakersów!

Stary, to wystarczy, żeby zrobić imprezę!

Zorganizuj sobie koc.

I możemy połączyć te dwie rzeczy, czyli zmienne i operacje arytmetyczne:

Masz 110 kawałków sera!

Masz 1050 paczek krakersów!

Stary, to wystarczy, żeby zrobić imprezę!

Zorganizuj sobie koc.

Zrób to sam

1. Przejrzyj jeszcze raz skrypt i nad każdą linią napisz komentarz wyjaśniający, co ona robi.
2. Zaczynij od dołu i przeczytaj każdą linię wstecz, wypowiadając na głos wszystkie ważne znaki.
3. Napisz co najmniej jeszcze jedną funkcję własnego projektu i uruchom ją na 10 różnych sposobów.

Typowe pytania

Jak może istnieć 10 różnych sposobów na uruchomienie funkcji? Możesz wierzyć lub nie, ale teoretycznie istnieje nieskończona liczba sposobów wywołania dowolnej funkcji. Sprawdź swoją kreatywność z funkcjami, zmiennymi i danych wejściowymi od użytkownika.

Czy istnieje jakiś sposób na przeanalizowanie działania tej funkcji, aby lepiej ją zrozumieć? Istnieje wiele różnych sposobów, ale spróbuj umieścić nad każdą linią komentarz opisujący, co ona robi. Kolejną sztuczką jest odczytanie kodu na głos. Jeszcze inną metodą jest wydrukowanie kodu na kartce i opatrzenie go rysunkami oraz komentarzami wyjaśniającymi, co się w nim dzieje.

Co zrobić, jeśli będę chciał zapytać użytkownika o liczbę kawałków sera i paczek krakersów? Musisz użyć `int()` do przekonwertowania tego, co otrzymasz z `input()`.

Czy utworzenie zmiennej `amount_of_cheese` (ilość sera) zmienia zmienną `cheese_count` (liczba kawałków sera) w funkcji? Nie, te zmienne są oddzielne i istnieją poza funkcją. Następnie są przekazywane do funkcji, a ich wersje tymczasowe są tworzone tylko w celu uruchomienia funkcji. Kiedy funkcja zakończy działanie, te tymczasowe zmienne znikają i wszystko działa dalej. Kontynuuj lekturę tej książki, a te rzeczy niedługo staną się bardziej zrozumiałe.

Czy błędem jest, kiedy mam zmienne globalne (takie jak `amount_of_cheese`) o tej samej nazwie, co zmienne funkcji? Tak, ponieważ wtedy nie jesteś do końca pewien, do której zmiennej się odnosisz. Czasami jednak używa się tej samej nazwy z konieczności lub przypadkiem. Po prostu unikaj tego, jeśli tylko możesz.

Czy istnieje ograniczenie liczby argumentów, które może posiadać funkcja? To zależy od wersji Pythona i komputera, na którym pracujesz, ale ten limit jest dość duży. Praktycznym ograniczeniem jest jednak około pięciu argumentów, zanim funkcja stanie się denerwująca w użyciu.

Czy można wywołać funkcję wewnątrz funkcji? Tak. Dalej w książce utworzysz grę, która będzie tak robić.

Funkcje i pliki

Pamiętaj o liście kontrolnej dla funkcji i wykonaj to ćwiczenie, zwracając szczególną uwagę na to, w jaki sposób funkcje i pliki mogą współpracować, aby robić różne użyteczne rzeczy.

ex20.py

```
1  from sys import argv
2
3  script, input_file = argv
4
5  def print_all(f):
6      print(f.read())
7
8  def rewind(f):
9      f.seek(0)
10
11  def print_a_line(line_count, f):
12      print(line_count, f.readline())
13
14  current_file = open(input_file)
15
16  print("Najpierw wydrukujmy cały plik:\n")
17
18  print_all(current_file)
19
20  print("Teraz przewińmy do tyłu, tak jak przewija się kasetę.")
21
22  rewind(current_file)
23
24  print("Wydrukujmy trzy linie:")
25
26  current_line = 1
27  print_a_line(current_line, current_file)
28
29  current_line = current_line + 1
30  print_a_line(current_line, current_file)
31
32  current_line = current_line + 1
33  print_a_line(current_line, current_file)
```

Zwróć szczególną uwagę na sposób przekazywania numeru bieżącej linii za każdym razem, gdy uruchamiamy funkcję `print_a_line`.

Co powinieneś zobaczyć

Ćwiczenie 20. — sesja

```
$ python3.6 ex20.py test.txt
Najpierw wydrukujmy cały plik:
```

```
To jest linia 1.
To jest linia 2.
To jest linia 3.
```

```
Teraz przewińmy do tyłu, tak jak przewija się kasetę.
Wydrukujmy trzy linie:
1 To jest linia 1.
2 To jest linia 2.
3 To jest linia 3.
```

Zrób to sam

1. Napisz komentarz dla każdej linii kodu, aby zrozumieć, co ona robi.
2. Za każdym razem, gdy uruchamiana jest funkcja `print_a_line`, przekazujesz do niej zmienną `current_line`. Wypisz wartości zmiennej `current_line` dla każdego wywołania funkcji i prześledź, w jaki sposób staje się ona `line_count` w `print_a_line`.
3. Znajdź każde miejsce, w którym używana jest funkcja, i sprawdź jej definicję (`def`), aby upewnić się, że podajesz właściwe argumenty.
4. Poszukaj w internecie, co robi funkcja `seek` dla pliku (`file`). Spróbuj poszukać informacji na ten temat w dokumentacji `pydoc file`. Następnie poczytaj dokumentację `pydoc file.seek`, aby zobaczyć, co robi `seek`.
5. Poszukaj informacji na temat skróconej notacji `+=` i napisz skrypt `od nowa` z jej wykorzystaniem.

Typowe pytania

Czym jest `f` w funkcji `print_all` i innych funkcjach? Jest to zmienna taka, jakie możesz zobaczyć w innych funkcjach w ćwiczeniu 18., ale tym razem jest to plik. Plik w Pythonie przypomina stary napęd taśmowy na komputerze typu mainframe lub odtwarzacz DVD. Ma „głowicę odczytu” i możesz „przeszukiwać” (ang. *seek*) plik za pomocą tej głowicy odczytu, ustawiając ją w różnych pozycjach, w których chcesz rozpocząć pracę. Za każdym razem, gdy robisz `f.seek(0)`, przesuwasz się do początku pliku. Gdy robisz `f.readline()`, odczytujesz linię z pliku i przesuwasz głowicę odczytu do pozycji bezpośrednio po znaku nowej linii (`\n`), który kończy tę linię. Wyjaśnię to dokładniej w kolejnych ćwiczeniach.

Dlaczego seek(0) nie ustawia wartości current_line na 0? Po pierwsze, funkcja seek() operuje na bajtach, a nie liniach. Kod seek(0) przenosi głowicę odczytu na zerowy bajt (pierwszy bajt) w pliku. Po drugie, current_line jest po prostu zmienną i nie ma żadnego rzeczywistego połączenia z plikiem. Inkrementujemy ją ręcznie.

Co to jest +=? Jest to tak zwana forma ściągnięta, używana na przykład w języku angielskim, gdzie możemy skrócić *it is* do *it's* lub *you are* do *you're*. W programowaniu jest to skrócona notacja dwóch operacji, takich jak = oraz +. Oznacza to, że $x = x + y$ jest tym samym, co $x += y$.

Skąd readline() „wie”, gdzie znajduje się każda linia? Wewnątrz funkcji readline() znajduje się kod, który skanuje każdy bajt pliku dopóki, dopóty nie napotka znaku \n. Wtedy wstrzymuje odczytywanie pliku, aby zwrócić to, co znalazł do tej pory. Plik f odpowiada za utrzymywanie aktualnej pozycji w pliku po każdym wywołaniu readline(), aby kontynuować czytanie każdej linii.

Dlaczego istnieją puste linie między wierszami w pliku? Funkcja readline() zwraca znak nowej linii (\n), który znajduje się w pliku na końcu danej linii. Aby uniknąć dodawania podwójnego \n do każdej linii, dodaj end = " " na końcu wywołania funkcji print.

Funkcje mogą coś zwracać

Używałeś już znaku równości (=) do nazywania zmiennych i przypisywania im liczb lub łańcuchów znaków. Teraz ponownie Cię zaskoczę i pokażę, jak używać = oraz nowego słowa Pythona `return` do przypisywania zmiennym **wartości z funkcji**. Będziesz musiał zwrócić szczególną uwagę na jedną rzecz, ale najpierw wpisz poniższy kod.

ex21.py

```

1  def add(a, b):
2      print(f"DODAWANIE {a} + {b}")
3      return a + b
4
5  def subtract(a, b):
6      print(f"ODEJMOWANIE {a} - {b}")
7      return a - b
8
9  def multiply(a, b):
10     print(f"MNOŻENIE {a} * {b}")
11     return a * b
12
13  def divide(a, b):
14     print(f"DZIELENIE {a} / {b}")
15     return a / b
16
17
18  print("Wykonajmy kilka operacji arytmetycznych jedynie za pomocą funkcji!")
19
20  age = add(30, 5)
21  height = subtract(78, 4)
22  weight = multiply(90, 2)
23  iq = divide(100, 2)
24
25  print(f"Wiek: {age}, Wzrost: {height}, Waga: {weight}, IQ: {iq}")
26
27
28  # To łamigłówka za dodatkowe punkty, ale i tak ją wpisz.
29  print("Oto zadanie.")
30
31  what = add(age, subtract(height, multiply(weight, divide(iq, 2))))
32
33  print("To daje: ", what, "Czy dałbyś radę obliczyć to ręcznie?")

```

Utworzyliśmy nasze własne funkcje matematyczne dla dodawania (`add`), odejmowania (`subtract`), mnożenia (`multiply`) i dzielenia (`divide`). Należy zwrócić uwagę na ostatnią linię, w której piszemy: `return a + b` (w funkcji `add`). W tej funkcji robimy następujące rzeczy.

1. Wywołujemy naszą funkcję z dwoma argumentami `a` i `b`.
2. Drukujemy, co robi nasza funkcja, w tym przypadku `DODAWANIE`.

3. Instruuujemy Pythona, żeby zrobił coś jakby trochę wstecz: zwracamy sumę dodawania $a + b$. Mógłbyś powiedzieć: „Dodaję a i b , a potem zwracam wynik”.
4. Python dodaje te dwie liczby. Następnie, gdy funkcja się zakończy, każda linia kodu, która ją uruchomi, będzie w stanie przypisać wynik operacji $a + b$ do zmiennej.

Podobnie jak wiele innych rzeczy w tej książce, powinieneś przeanalizować to naprawdę po woli, rozłożyć na części i spróbować prześledzić, co się dzieje. Na koniec masz zadanie za dodatkowe punkty — rozwiązując je, nauczysz się czegoś fajnego.

Co powinieneś zobaczyć

Ćwiczenie 21. — sesja

```
$ python3.6 ex21.py
Wykonajmy kilka operacji arytmetycznych jedynie za pomocą funkcji!
DODAWANIE 30 + 5
ODEJMOWANIE 78 - 4
MNOŻENIE 90 * 2
DZIELENIE 100 / 2
Wiek: 35, Wzrost: 74, Waga: 180, IQ: 50.0
Oto zadanie.
DZIELENIE 50.0 / 2
MNOŻENIE 180 * 25.0
ODEJMOWANIE 74 - 4500.0
DODAWANIE 35 + -4426.0
To daje: -4391.0 Czy dałbyś radę obliczyć to ręcznie?
```

Zrób to sam

1. Jeśli nie jesteś do końca pewien, co robi `return`, spróbuj napisać kilka własnych funkcji, które zwracają jakieś wartości. Możesz zwracać cokolwiek, co umieścisz po prawej stronie znaku `=`.
2. Na końcu skryptu jest łamigłówka. Biorę wartość zwracaną z jednej funkcji i używam jej jako argumentu dla drugiej funkcji. Robię to w łańcuchu, więc w pewien sposób tworzę formułę, korzystając z funkcji. Wygląda to naprawdę dziwnie, ale jeśli uruchomisz skrypt, zobaczysz wyniki. Powinieneś spróbować znaleźć normalną formułę, która odtworzy ten sam zestaw operacji.
3. Kiedy już wypracujesz formułę dla łamigłówki, pobaw się nią, modyfikując fragmenty funkcji. Spróbuj zmienić coś celowo w taki sposób, aby uzyskać inną wartość.
4. Wykonaj ćwiczenie odwrotne. Napisz prostą formułę i użyj funkcji w ten sam sposób, aby ją obliczyć.

Wykonując to ćwiczenie, naprawdę będziesz musiał wyteńczyć umysł, ale nie spiesz się i potraktuj to jak małą grę. Dzięki rozwiązywaniu takich łamigłówek programowanie staje się zabawą, więc będę zadawał Ci więcej takich zadań.

Typowe pytania

Dlaczego Python drukuje formułę albo funkcję „wstecz”? Naprawdę to nie jest wstecz, tylko „na lewą stronę”. Gdy zaczniesz rozkładać funkcję na osobne formuły i wywołania funkcji, zobaczysz, jak to działa. Postaraj się zrozumieć to w kategoriach „na lewą stronę” zamiast „wstecz”.

Jak mogę użyć `input()` do wprowadzenia własnych wartości? Pamiętasz `int(input())`? Problem z takim rozwiązaniem polega na tym, że nie można wprowadzać wartości zmiennoprzecinkowych, więc spróbuj zamiast tego użyć również `float(input())`.

Co masz na myśli, mówiąc „napisz formułę”? Na początek spróbuj $24 + 34 / 100 - 1023$. Przekonwertuj to na użycie funkcji. Teraz wymyśl własne podobne równanie matematyczne i użyj zmiennych, żeby bardziej przypominało formułę.

Czego nauczyłeś się do tej pory?

W tym ćwiczeniu i następnym nie będzie żadnego kodu, więc nie będzie też podrozdziałów „Co powinienś zobaczyć” i „Zrób to sam”. W rzeczywistości to ćwiczenie to jedno wielkie „Zrób to sam”. Zrobisz w nim przegląd tego, czego dotychczas się nauczyłeś.

Najpierw wróć do każdego wykonanego do tej pory ćwiczenia i spisz wszystkie słowa oraz symbole (inaczej znaki), których użyłeś. Upewnij się, że lista symboli jest kompletna.

Przy każdym słowie lub symbolu zapisz ich nazwę i opisz, co one robią. Jeśli nie możesz znaleźć nazwy jakiegoś symbolu w tej książce, poszukaj jej w internecie. Gdy nie wiesz, co robią dane słowo lub dany symbol, poczytaj o nich ponownie i spróbuj użyć w jakimś kodzie.

Możesz natknąć się na kilka rzeczy, których nie znasz lub nie możesz wyszukać, więc po prostu zatrzymaj je na liście i bądź gotowy, żeby sprawdzić później.

Gdy będziesz miał już gotową listę, poświęć kilka dni na sporządzanie jej od nowa i sprawdzanie, czy jest poprawna. Może to być nużące, ale przebrnij przez to i naucz się jej na pamięć.

Po zapamiętaniu pozycji z listy wraz z opisami przejdź na wyższy poziom trudności i wypisz z *pamięci* wszystkie symbole, ich nazwy i funkcjonalności. Jeśli nie będziesz w stanie czegoś sobie przypomnieć, wróć do tego i jeszcze raz spróbuj zapamiętać.

OSTRZEŻENIE! Podczas wykonywania tego ćwiczenia pamiętaj o jednej ważnej rzeczy: „Porażka nie istnieje, są tylko próby”.

Miej świadomość, czego się uczysz

Gdy wykonujesz nudne, bezmyślne ćwiczenie pamięciowe, takie jak to, powinienś wiedzieć, dlaczego to robisz. Wiedza ta pomaga skupić się na celu, do którego dążysz.

W tym ćwiczeniu uczysz się nazw symboli, aby łatwiej czytać kod źródłowy. Przypomina to naukę alfabetu i podstawowych słówek w dowolnym obcym języku, z tym wyjątkiem, że alfabet Pythona zawiera dodatkowe symbole, których możesz nie znać.

Po prostu się nie spiesz i zbytnio nie forsuj. Najlepiej rób sobie przerwy co 15 minut — zachowanie świeżego umysłu pomoże uczyć się szybciej przy mniejszym poziomie frustracji.

Łańcuchy znaków, bajty i kodowanie znaków

Aby wykonać to ćwiczenie, musisz pobrać z serwera FTP wydawnictwa Helion (<ftp://ftp.helion.pl/przyklady/pyt3pw.zip>) przygotowany przeze mnie plik *languages.txt*. Plik zawiera listę różnych języków i ma na celu zademonstrowanie kilku interesujących koncepcji.

1. W jaki sposób nowoczesne komputery zapisują języki w celu ich wyświetlania i przetwarzania oraz dlaczego Python 3 nazywa to łańcuchem znaków, czyli typem `string`?
2. Dlaczego trzeba „kodować” łańcuchy znaków Pythona do postaci bajtów, czyli typu `bytes`, a następnie je „dekodować”?
3. Jak radzić sobie z błędami w obsłudze łańcuchów znaków i bajtów?
4. Jak czytać kod i rozumieć jego znaczenie, nawet jeśli nigdy wcześniej go nie widziałeś?

Oprócz tego zapoznasz się również z krótkim opisem instrukcji `if` Pythona 3. Nie musisz od razu opanowywać tego kodu ani rozumieć pojęć. Te rzeczy będą się pojawiać również w kolejnych ćwiczeniach. Na razie otrzymasz przedsmak tego, czego będziesz się uczył później, a Twoim zadaniem jest zapoznanie się z wymienionymi powyżej zagadnieniami.

OSTRZEŻENIE! To ćwiczenie jest trudne! Jest w nim dużo informacji, które musisz przyswoić, a omawiane zagadnienia związane są z podstawami działania komputerów. To ćwiczenie jest trudne, ponieważ łańcuchy znaków Pythona są złożone i niełatwe w użyciu. Zalecam, abyś wykonywał to ćwiczenie *bardzo, bardzo powoli*. Zapisz każde słowo, którego nie rozumiesz, i sprawdź je lub poszukaj informacji w internecie. Jeśli musisz, czytaj po jednym akapicie na raz. Wykonując to ćwiczenie, możesz jednocześnie zająć się kolejnymi ćwiczeniami, aby nie utknąć w tym miejscu. Po prostu wykonuj je skrupulatnie kawałek po kawałku tak długo, jak to będzie konieczne.

Wstępne badanie kodu

Nauczę Cię, jak zbadać fragment kodu, aby ujawnić jego sekrety. Aby ten kod zadziałał, będziesz potrzebował pliku *languages.txt*, więc najpierw go pobierz. Plik *languages.txt* zawiera po prostu listę języków zakodowaną w systemie UTF-8.

ex23.py

```
1 import sys
2 script, encoding, error = sys.argv
3
```

```

4
5  def main(language_file, encoding, errors):
6      line = language_file.readline()
7
8      if line:
9          print_line(line, encoding, errors)
10         return main(language_file, encoding, errors)
11
12
13  def print_line(line, encoding, errors):
14      next_lang = line.strip()
15      raw_bytes = next_lang.encode(encoding, errors=errors)
16      cooked_string = raw_bytes.decode(encoding, errors=errors)
17
18      print(raw_bytes, "<====>", cooked_string)
19
20
21  languages = open("languages.txt", encoding="utf-8")
22
23  main(languages, encoding, error)

```

W tych przykładach użyłem kodowań utf-8, utf-16 i big5 w celu zademonstrowania konwersji i różnych rodzajów błędów, jakie możesz otrzymać. W Pythonie 3 każde z tych kodowań jest nazywane „kodekiem”, ale nasz parametr nazwałem encoding (kodowanie). Omówię pokrótce, co oznaczają dane wyjściowe. Wystarczy Ci ogólne pojęcie, jak to działa, żebyśmy mogli o tym porozmawiać.

Po uruchomieniu tego skryptu kilka razy przejrzyj listę symboli i postaraj się odgadnąć, do czego służą. Zanotuj swoje domysły i spróbuj wyszukać te symbole w internecie, aby sprawdzić, czy możesz potwierdzić swoje hipotezy. Nie przejmuj się, jeśli nie masz pojęcia, jak je wyszukiwać. Po prostu próbuj.

Przełączniki, konwencje i rodzaje kodowania

Zanim będę mógł zacząć objaśniać ten kod, musisz nauczyć się podstaw dotyczących sposobu przechowywania danych w komputerze. Współczesne komputery są niewiarygodnie skomplikowane, ale zasadniczo są jak ogromna tablica przełączników elektrycznych. Komputery wykorzystują energię elektryczną do włączania i wyłączania przełączników. Przełączniki mogą reprezentować cyfrę 1 dla pozycji włączonej lub cyfrę 0 dla pozycji wyłączonej. W dawnych czasach istniały różne rodzaje dziwnych komputerów, które robiły jeszcze inne rzeczy poza ustawianiem 1 lub 0, ale obecnie stosuje się wyłącznie systemy zero-jedynkowe. Jedynka reprezentuje energię, elektryczność, włączone, zasilanie, materię. Zero reprezentuje wyłączone, zrobione, usunięte, brak zasilania, brak energii. Te zera i jedynki nazywamy bitami.

Jednak komputer, który pozwalałby pracować tylko z cyframi 1 i 0, byłby wyjątkowo nieefektywny i niesamowicie irytujący. Komputery używają tych jedynek i zer do kodowania większych liczb. W formacie little endian komputer będzie używał ośmiu z tych jedynek i zer do zakodowania 256 liczb (0–255). Jednak co oznacza „kodowanie”? To nic więcej jak tylko uzgodniony standard, określający, w jaki sposób sekwencja bitów powinna reprezentować liczbę. Jest to ustalona konwencja, zgodnie z którą 00000000 reprezentuje liczbę 0, 11111111 to 255, a 00001111 ma wartość 15. W początkowej fazie rozwoju informatyki toczyły się nawet zaciekle spory dotyczące samej tylko kolejności tych bitów, ponieważ były to po prostu konwencje, na które wszyscy musieli się zgodzić.

Dziś „bajtem” nazywamy sekwencję 8 bitów (jedynek i zer). Dawniej każdy miał własną koncepcję dotyczącą bajta, więc nadal będziesz spotykał ludzi, którzy myślą, że ten termin powinien być elastyczny i obejmować sekwencje 9, 7 lub 6 bitów — teraz jednak mówimy, że bajt ma 8 bitów. To jest nasza konwencja i ta konwencja definiuje metodę kodowania dla bajta. Istnieją dalsze konwencje kodowania dużych liczb za pomocą 16, 32, 64 lub jeszcze większej liczby bitów, jeśli wejdziemy w naprawdę wielką matematykę. Są całe grupy standardów, których twórcy i zwolennicy spierają się o te konwencje, a następnie implementują je jako kodowanie, które ostatecznie włącza i wyłącza przełączniki.

Gdy masz już bajty, możesz zacząć przechowywać i wyświetlać tekst, decydując się na kolejną konwencję mapowania liczb na litery. We wczesnej fazie rozwoju informatyki istniało wiele konwencji mapujących 8 lub 7 bitów (mniej więcej) na listy znaków przechowywanych w komputerze. Najpopularniejszą konwencją okazał się ostatecznie standard **ASCII** (ang. *American Standard Code for Information Interchange*). Standard ten mapuje liczby na litery.

Litera „Z” to na przykład liczba 90 (w bitach to 1011010), która jest mapowana na tabelę ASCII wewnątrz komputera.

Możesz wypróbować to teraz w Pythonie.

```
>>> 0b1011010
90
>>> ord('Z')
90
>>> chr(90)
'Z'
>>>
```

Najpierw wpisuję liczbę 90 w postaci binarnej, następnie otrzymuję liczbę na podstawie wpisanej litery „Z”, a potem konwertuję liczbę na literę „Z”. Nie musisz starać się tego zapamiętać. Myślę, że musiałem to zrobić może ze dwa razy przez cały czas korzystania z Pythona.

Kiedy mamy już konwencję ASCII do kodowania znaków za pomocą 8 bitów (bajta), możemy łączyć je w „łańcuchy znaków”, aby tworzyć słowa. Jeśli chcę napisać moje imię i nazwisko, „Zed A. Shaw”, używam po prostu sekwencji bajtów: [90, 101, 100, 32, 65, 46, 32, 83, 104, 97, 119]. Większość wczesnych tekstów na komputerach była sekwencją bajtów przechowywanych w pamięci, której komputer używał do wyświetlania tekstu użytkownikowi. Powtórzę jeszcze raz, że jest to tylko sekwencja konwencji, które włączają i wyłączają przełączniki.

Problem ze standardem ASCII polega na tym, że koduje on tylko język angielski i może kilka innych podobnych języków. Pamiętaj, że bajt może pomieścić 256 liczb (0 – 255 lub 00000000 – 11111111). Okazuje się, że w różnych językach świata używanych jest dużo więcej znaków niż 256. Różne kraje stworzyły własne konwencje kodowania dla swoich języków, które przeważnie działały prawidłowo, ale większość kodowań mogła obsługiwać tylko jeden język. Oznaczało to, że jeśli chciałeś umieścić tytuł angielskiej książki w środku zdania w języku tajskim, miałeś problem. Potrzebowałeś jednego kodowania dla języka tajskiego i jednego dla języka angielskiego.

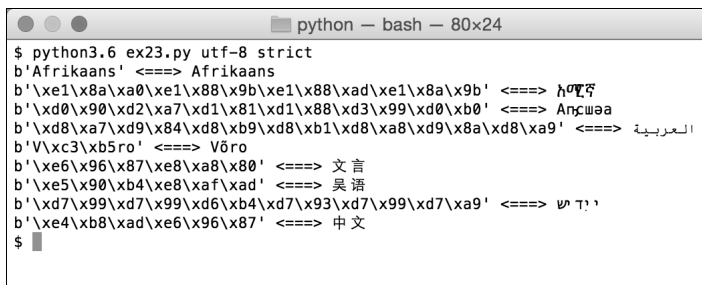
Aby rozwiązać ten problem, grupa ludzi opracowała standard Unicode. Jego nazwa brzmi podobnie do angielskiego słowa *encode* (oznaczającego kodowanie) i ma on być „uniwersalnym kodowaniem” wszystkich ludzkich języków. Rozwiązanie, które zapewnia Unicode, jest podobne do tabeli ASCII, ale jest to ogromna tabela. Do zakodowania znaku Unicode możesz użyć 32 bitów, a to więcej znaków niż moglibyśmy gdziekolwiek znaleźć. Liczba 32-bitowa oznacza, że możemy przechowywać $4\,294\,967\,295$ znaków (2^{32}), co jest wystarczającą przestrzenią dla każdego możliwego ludzkiego języka i prawdopodobnie także wielu języków pozaziemskich. Obecnie używamy tej dodatkowej przestrzeni do różnych ważnych rzeczy, takich jak emotikony kupy i uśmiechy.

Mamy już konwencję do kodowania dowolnych znaków, ale 32 bity to 4 bajty ($32/8 = 4$), co oznacza, że w większości tekstów, które chcemy zakodować, marnuje się wiele przestrzeni. Możemy również użyć 16 bitów (2 bajtów), ale nadal w większości tekstów będziemy marnować miejsce. Rozwiązaniem jest użycie sprytnej konwencji do kodowania najczęstszych znaków za pomocą 8 bitów i „ucieczka” w większe liczby, gdy trzeba zakodować więcej znaków. Oznacza to, że mamy jeszcze jedną konwencję, która jest tylko kodowaniem z kompresją, dzięki czemu dla większości popularnych znaków możemy używać 8 bitów, a następnie uciekać w 16 bitów lub 32 bity, jeśli trzeba.

Konwencja do kodowania tekstu w Pythonie nosi nazwę **UTF-8** (ang. *Unicode Transformation Format — 8-bit*). Jest to konwencja do kodowania znaków Unicode w sekwencji bajtów, które są sekwencjami bitów, które są z kolei sekwencjami przełączników włączone-wyłączone. Możesz używać również innych konwencji (kodowań), ale standardem jest UTF-8.

Analizujemy dane wyjściowe

Możemy teraz przyjrzeć się danym wyjściowym z pokazanych wcześniej poleceń. Przeanalizujmy tylko to pierwsze polecenie i kilka początkowych linii danych wyjściowych.



```
python — bash — 80x24
$ python3.6 ex23.py utf-8 strict
b'Afrikaans' <====> Afrikaans
b'\xe1\x8a\xa0\xe1\x88\x9b\xe1\x88\xad\xe1\x8a\x9b' <====> አሞሽ
b'\xd0\x90\xd2\xa7\xd1\x81\xd1\x88\xd3\x99\xd0\xb0' <====> Ангшөө
b'\xd8\xa7\xd9\x84\xd8\xb9\xd8\xb1\xd8\xa8\xd9\x8a\xd8\xa9' <====> العربية
b'\xc3\xb5ro' <====> Võro
b'\xe6\x96\x87\xe8\xa8\x80' <====> 文言
b'\xe5\x90\xb4\xe8\xaf\xad' <====> 吳語
b'\xd7\x99\xd7\x99\xd6\xb4\xd7\x93\xd7\x99\xd7\xa9' <====> 𐤒 𐤓 𐤔
b'\xe4\xb8\xad\xe6\x96\x87' <====> 中文
$
```

Skrypt `ex23.py` pobiera bajty zapisane wewnątrz ciągu bajtów `b' '` i konwertuje je na określone kodowanie: UTF-8 lub inne. Po lewej stronie znajdują się liczby dla każdego bajta kodowania UTF-8 (pokazane w systemie szesnastkowym), a po prawej znak wyjściowy jako rzeczywisty kod UTF-8. Można powiedzieć, że lewa strona `<====>` to bajty numeryczne Pythona lub inaczej „surowe” bajty używane przez Pythona do przechowywania łańcucha znaków. Aby wskazać Pythonowi te bajty, określa się je za pomocą `b' '`. Te surowe bajty są następnie wyświetlane w postaci „przetworzonej” po prawej stronie, żeby można było zobaczyć w terminalu rzeczywiste znaki.

Analizujemy kod

Wiemy już, czym są łańcuchy znaków i sekwencje bajtów. W Pythonie string jest zakodowaną w UTF-8 sekwencją znaków, służącą do wyświetlania tekstu lub pracy z nim. Natomiast bytes to „surowa” sekwencja bajtów, której Python używa do przechowywania tego obiektu string UTF-8; zaczyna się ona od `b' '`, aby poinformować Pythona, że pracujemy z surowymi bajtami. Wszystko opiera się na konwencjach dotyczących sposobu, w jaki Python pracuje z tekstem. Oto sesja Pythona, w której koduję łańcuchy znaków i dekoduję bajty.

Wystarczy zapamiętać, że jeśli masz surowe bajty (bytes), musisz użyć `.decode()`, aby uzyskać łańcuch znaków (string). Do surowych bajtów nie stosuje się żadnej konwencji. Są to po prostu ciągi bajtów, które nie mają poza liczbami innego znaczenia, więc musisz powiedzieć Pythonowi, aby „zdekodował to na łańcuch znaków UTF”. Jeśli masz string i chcesz go wysłać, zapisać, udostępnić lub wykonać na nim jakąś inną operację, zwykle będzie to działać,

```
python — bash — 82x34
$ python3.6
Python 3.6.0 (default, Feb  2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>> raw_bytes = b'\xe6\x96\x87\xe8\xa8\x80'
>>> utf_string = "文言"
>>> raw_bytes.decode()
'文言'
>>> utf_string.encode()
b'\xe6\x96\x87\xe8\xa8\x80'
>>> raw_bytes == utf_string.encode()
True
>>> utf_string == raw_bytes.decode()
True
>>>
>>> quit()
$
```

ale czasami Python rzuci błąd z informacją, że nie „wie”, jak to „zakodować”. Przypomnę, że Python zna swoje wewnętrzne konwencje, ale nie ma pojęcia, której konwencji potrzebujesz. W takim przypadku musisz użyć `.encode()`, aby otrzymać potrzebne bajty.

W języku angielskim istnieje pewna mnemotechnika pozwalająca to zapamiętać (choć za każdym i tak sprawdzam, co powinienem zrobić): **DBES** (ang. *Decode Bytes, Encode Strings*), co oznacza „dekoduj bajty, koduj łańcuchy znaków”. Powtarzam sobie to w myślach (wymawiając „dibes”), kiedy muszę konwertować bytes i string. Kiedy masz bytes i potrzebujesz string, dekoduj bajty. Kiedy masz string i potrzebujesz bytes, koduj łańcuchy znaków.

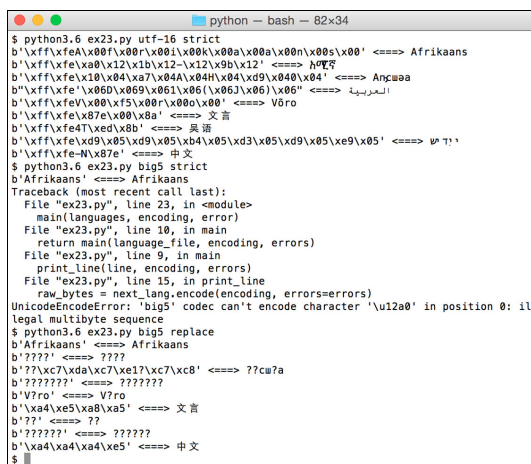
Mając to na uwadze, przeanalizujmy kod z pliku `ex23.py` linia po linii.

1. – 2. Zaczynam od standardowej obsługi argumentów wiersza poleceń, którą już znasz.
5. Rozpaczynam główną część tego kodu w funkcji wygodnie nazywanej `main`. Będzie ona wywoływana pod koniec tego skryptu, aby rozpocząć jego działanie.
6. Pierwszą rzeczą, którą robi ta funkcja, jest odczytanie jednej linii z podanego jej pliku z językami. Robiłeś to już wcześniej, więc nie ma tu niczego nowego. Używamy po prostu `readline`, tak jak poprzednio, gdy mieliśmy do czynienia z plikami tekstowymi.
8. Tutaj używam czegoś nowego. Będziesz się o tym uczył w drugiej połowie książki, więc potraktuj to jako zapowiedź ciekawych rzeczy. Jest to instrukcja `if`, która pozwala podejmować określone decyzje w kodzie Pythona. Możesz sprawdzić prawdziwość zmiennej i na podstawie tego warunku logicznego uruchomić fragment kodu lub nie. W tym przypadku testuję, czy `line` ma jakąś wartość. Funkcja `readline` zwróci pusty łańcuch znaków, gdy dojdzie do końca pliku, a `if line` po prostu sprawdza, czy zwrócony został ten pusty łańcuch znaków. Dopóki `readline` zwraca jakąś wartość, będzie to prawda i będzie uruchamiany znajdujący się *poniżej* kod (wcięte linie 9. i 10.). Gdy będzie to fałsz, Python pominie linie 9. i 10.

9. Wywołuję osobną funkcję, aby wykonać faktyczne drukowanie tej linii. To upraszcza mój kod i ułatwia mi jego zrozumienie. Jeśli chcę się dowiedzieć, co robi ta funkcja, mogę do niej przeskoczyć i ją przeanalizować. Gdy już wiem, co robi `print_line`, mogę skojarzyć to z nazwą tej funkcji i zapomnieć o szczegółach.
10. Napisałem tutaj niewielki, ale potężny kawałek magii. Wywołuję ponownie `main` wewnątrz `main`. Właściwie to nie jest magia, ponieważ w programowaniu naprawdę nie ma niczego magicznego. Dostępne są wszystkie potrzebne informacje. Wygląda na to, że wywołuję tę funkcję *wewnątrz* samej siebie, co chyba powinno być niedozwolone. Zadaż sobie pytanie, dlaczego miałoby to być niedozwolone? Nie ma żadnego technicznego powodu, dla którego nie mógłbym tam wywołać dowolnej funkcji, nawet tej funkcji `main`. Jeśli funkcja jest po prostu przeskoczeniem do góry skryptu, gdzie nazwałem ją `main`, wtedy wywołanie tej funkcji na końcu jej samej będzie... przeskoczeniem do góry skryptu i uruchomieniem jej ponownie. To by ją zapętlilo. Teraz spójrz na linię 8., a zobaczysz instrukcję `if`, która zapobiega zapętleniu tej funkcji w nieskończoność. Przystudiuj to uważnie, ponieważ to ważna koncepcja, ale nie przejmuj się, jeśli nie od razu ją zrozumiesz.
13. Rozpaczam definicję funkcji `print_line`, która wykonuje faktyczne kodowanie każdej linii z pliku *languages.txt*.
14. Jest to proste pozbywanie się końcowego `\n` z łańcucha znaków `line`.
15. Wreszcie pobieram język z otrzymanego pliku *languages.txt* i koduję go do postaci surowych bajtów. Pamiętaj o mnemotechnice DBES: dekoduj bajty, koduj łańcuchy znaków. Zmienna `next_lang` jest łańcuchem znaków, więc aby uzyskać surowe bajty, muszę wywołać na niej `.encode()` w celu zakodowania łańcucha znaków. Do funkcji `encode()` przekazuję rodzaj kodowania i obsługę błędów.
16. Tworząc zmienną `cooked_string` z `raw_bytes`, wykonuję dodatkowy krok pokazujący odwrotność linii 15. Pamiętaj, że DBES mówi o dekodowaniu bajtów, a `raw_bytes` to `bytes`, więc wywołuję na niej `.decode()`, aby uzyskać `string` Pythona. Ten łańcuch znaków powinien być taki sam jak zmienna `next_lang`.
18. Po prostu drukuje obie zmienne, aby pokazać, jak wyglądają.
21. Skończyłem definiowanie funkcji, więc teraz chcę otworzyć plik *languages.txt*.
23. Koniec skryptu uruchamia po prostu funkcję `main` ze wszystkimi poprawnymi parametrami, aby program zaczął działać i rozpoczęła się pętla. Pamiętaj, że następnie powoduje to przeskoczenie do miejsca zdefiniowania funkcji `main` w linii 5., a w linii 10. funkcja `main` jest wywoływana ponownie, co domyka pętlę. Instrukcja `if line`: w linii 8. zapobiega zapętleniu w nieskończoność.

Bawimy się kodowaniem

Możemy teraz użyć naszego małego skryptu do zbadania innych kodowań. Na poniższym rysunku widać, jak bawię się różnymi kodowaniami i sprawdzam, co zadziała. Najpierw wykonuję proste kodowanie UTF-16, żebyś mógł zobaczyć, jak zmienia się ono w porównaniu do UTF-8. Możesz także użyć UTF-32, aby zobaczyć, że jest jeszcze dłuższe, i uzyskać wyobrażenie przestrzeni zaoszczędzonej za pomocą UTF-8. Potem próbuję kodowania Big5 i możesz zobaczyć, że Python w ogóle go *nie lubi*. Rzuca błąd, że Big5 nie może zakodować niektórych znaków w pozycji 0 (co jest superpomocne). Jednym z rozwiązań jest powiadzenie Pythonowi, aby „zastąpił” (replace) wszystkie złe znaki dla kodowania Big5. Robię to w następnym poleceniu i — jak widać — Python wstawia ? wszędzie tam, gdzie napotyka znak, który nie pasuje do systemu kodowania Big5.



```
python -- bash -- 82x34
$ python3.6 ex23.py utf-16 strict
b'\xff\xfe\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' <==== Afrikaans
b'\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' <==== 非洲
b'\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' <==== Анцааа
b'\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' <==== العربية
b'\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' <==== Võro
b'\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' <==== 文盲
b'\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' <==== 文盲
b'\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' <==== 文盲
b'\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' <==== 中文
$ python3.6 ex23.py big5 strict
b'Afrikaans' <==== Afrikaans
Traceback (most recent call last):
  File "ex23.py", line 23, in <module>
    main(languages, encoding, error)
  File "ex23.py", line 10, in main
    return main(language_file, encoding, errors)
  File "ex23.py", line 9, in main
    print_line(line, encoding, errors)
  File "ex23.py", line 15, in print_line
    raw_bytes = next_lang.encode(encoding, errors=errors)
UnicodeEncodeError: 'big5' codec can't encode character '\u0627' in position 0: il
legal multi-byte sequence
$ python3.6 ex23.py big5 replace
b'Afrikaans' <==== Afrikaans
b'?????' <==== ????
b'??x7?x8a?x7?x8a?x7?x8a?x7?x8a?' <==== ??cw7a
b'?????????' <==== ????????
b'V7ro?' <==== V7ro
b'\xa4\xe5\xa8\xa5' <==== 文盲
b'???' <==== ??
b'?????' <==== ??????
b'\xa4\xa4\xa4\xe5' <==== 中文
$
```

Popsuj kod

Oto kilka prostych pomysłów.

1. Znajdź łańcuchy tekstu zakodowane w innych systemach kodowania i umieść je w pliku *languages.txt*, aby zobaczyć, jak uruchomienie skryptu wywołuje błąd.
2. Zobacz, co się stanie, gdy jako argument podasz kodowanie, które nie istnieje.
3. Dodatkowe wyzwanie: przepisz plik *languages.txt*, używając ciągów bajtów `b'` zamiast łańcuchów znaków UTF-8, co w efekcie odwróci działanie skryptu.
4. Jeśli uda Ci się to zrobić, możesz także *popsuć* te ciągi bajtów, usuwając niektóre bajty, aby zobaczyć, co się stanie. Jak dużo trzeba usunąć, aby spowodować błąd Pythona? Ile możesz usunąć, żeby uszkodzić wynikowy łańcuch znaków, ale nie wywołać błędu systemu dekodowania Pythona?
5. Użyj tego, czego nauczyłeś się z punktu 4., aby sprawdzić, czy potrafisz uszkodzić plik. Jakie błędy otrzymałeś? Jak dużo szkód możesz wyrządzić, żeby plik nie spowodował błędu systemu dekodowania Pythona?

Więcej praktyki

Zbliżasz się do końca tej części książki. Powinieneś już mieć wystarczająco dużo Pythona „za paznokciami”, aby przejść do nauki o tym, jak naprawdę działa programowanie, ale najpierw powinieneś wykonać jeszcze trochę ćwiczeń. To ćwiczenie jest dłuższe i polega na budowaniu wytrzymałości. Następne ćwiczenie będzie podobne. Wykonaj je, zrób wszystko, tak jak należy, i sprawdź swój kod.

ex24.py

```

1  print("Przećwiczmy wszystko.")
2  print('Musisz poćwiczyć sekwencje ucieczki ze znakiem \\, które wstawiają:')
3  print('\n nowe linie oraz \t tabulatory.')
4
5  poem = """
6  \tTen piękny świat
7  z tak mocno osadzoną logiką
8  nie potrafi dostrzec \n potrzeby miłości
9  ani pojąć pasji płynącej z przeczcucia
10 i wymaga wyjaśnienia
11 \n\t\tale żadnego nie ma.
12 """
13
14 print("-----")
15 print(poem)
16 print("-----")
17
18
19 five = 10 - 2 + 3 - 6
20 print(f"To powinno wynosić pięć: {five}")
21
22 def secret_formula(started):
23     jelly_beans = started * 500
24     jars = jelly_beans / 1000
25     crates = jars / 100
26     return jelly_beans, jars, crates
27
28
29 start_point = 10000
30 beans, jars, crates = secret_formula(start_point)
31
32 # pamiętaj, że jest to kolejny sposób formatowania łańcucha znaków
33 print("Zaczynamy od wartości początkowej: {}".format(start_point))
34 # działa to podobnie do łańcucha znaków f"
35 print(f"To nam da {beans} żelek, {jars} słoików oraz {crates} skrzyń.")
36
37 start_point = start_point / 10
38
39 print("Możemy również zrobić to w ten sposób:")
40 formula = secret_formula(start_point)

```

```
41 # jest to prosty sposób zastosowania listy do sformatowanego łańcucha znaków
42 print("To nam da {} żelek, {} słoików oraz {} skrzyń.".format(*formula))
```

Co powinieneś zobaczyć

Ćwiczenie 24. — sesja

```
$ python3.6 ex24.py
Przećwiczmy wszystko.
Musisz poćwiczyć sekwencje ucieczki ze znakiem \, które wstawiają:
```

```
nowe linie oraz      tabulatory.
-----
```

```

        Ten piękny świat
z tak mocno osadzoną logiką
nie potrafi dostrzec
potrzeby miłości
ani pojąć pasji płynącej z przeczucia
i wymaga wyjaśnienia
```

```

        ale żadnego nie ma.
```

```
-----
To powinno wynosić pięć: 5
Zaczynamy od wartości początkowej: 10000
To nam da 5000000 żelek, 5000.0 słoików oraz 50.0 skrzyń.
Możemy również zrobić to w ten sposób:
To nam da 500000.0 żelek, 500.0 słoików oraz 5.0 skrzyń.
```

Zrób to sam

1. Pamiętaj o wykonaniu sprawdzania kodu: przeczytaj kod wstecz, czytaj go na głos i umieść komentarze przy wszystkich niejasnych fragmentach.
2. Celowo popsuj skrypt, a następnie uruchom go, aby zobaczyć, jakie otrzymasz błędy. Upewnij się, że potrafisz to naprawić.

Typowe pytania

Dlaczego nazywasz zmienną `jelly_beans`, ale później używasz nazwy `beans`? Jest to związane ze sposobem działania funkcji. Pamiętaj, że wewnątrz funkcji zmienna jest tymczasowa. Gdy ją zwracasz, możesz ją przypisać do jakiejś zmiennej do późniejszego użycia. Utworzyłem po prostu nową zmienną o nazwie `beans` do przechowywania zwracanej wartości.

Co masz na myśli, mówiąc o czytaniu kodu wstecz? Zaczynij od ostatniej linii. Porównaj tę linię w Twoim pliku z tą samą linią w moim. Jeśli obie linie są dokładnie takie same, przejdź do następnej. Powtarzaj te czynności, aż dojdiesz do pierwszej linii pliku.

Kto napisał ten wiersz? Ja. Nie wszystkie moje wiersze są do bani.

Jeszcze więcej praktyki

Zrobimy jeszcze więcej ćwiczeń obejmujących funkcje i zmienne, aby upewnić się, że dobrze je znasz. Ćwiczenie powinno być proste do wpisania, przeanalizowania i zrozumienia.

Jednak to ćwiczenie jest nieco inne. Nie będziesz go uruchamiał. Zamiast tego *zaimportujesz* je do Pythona i uruchomisz funkcje samodzielnie.

ex25.py

```
1  def break_words(stuff):
2      """Ta funkcja rozбивa zdanie na słowa."""
3      words = stuff.split(' ')
4      return words
5
6  def sort_words(words):
7      """Sortuje słowa."""
8      return sorted(words)
9
10 def print_first_word(words):
11     """Drukuje pierwsze słowo i usuwa je ze zdania."""
12     word = words.pop(0)
13     print(word)
14
15 def print_last_word(words):
16     """Drukuje ostatnie słowo i usuwa je ze zdania."""
17     word = words.pop(-1)
18     print(word)
19
20 def sort_sentence(sentence):
21     """Pobiera pełne zdanie i zwraca posortowane słowa."""
22     words = break_words(sentence)
23     return sort_words(words)
24
25 def print_first_and_last(sentence):
26     """Drukuje pierwsze i ostatnie słowo zdania."""
27     words = break_words(sentence)
28     print_first_word(words)
29     print_last_word(words)
30
31 def print_first_and_last_sorted(sentence):
32     """Sortuje słowa, a następnie drukuje pierwsze i ostatnie."""
33     words = sort_sentence(sentence)
34     print_first_word(words)
35     print_last_word(words)
```

Najpierw uruchom ten plik za pomocą polecenia `python3.6 ex25.py`, aby znaleźć błędy, które popełniłeś. Po znalezieniu wszystkich błędów możesz je naprawić, a następnie przejść do podrozdziału „Co powinienś zobaczyć”, żeby ukończyć to ćwiczenie.

Co powinieneś zobaczyć

W tym ćwiczeniu będziemy współpracować z plikiem `ex25.py` wewnątrz interpretera `python3.6`, którego od czasu do czasu używałeś do wykonywania obliczeń. Uruchom polecenie `python3.6` z terminala w ten sposób:

```
$ python3.6
Python 3.6.0rc2 (v3.6.0rc2:800a67f7806d, Dec 16 2016, 14:12:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Twoje dane wyjściowe powinny wyglądać tak jak moje, a po znaku `>` (zwanym znakiem zachęty) możesz wpisywać kod Pythona i natychmiast go uruchamiać. Chcę, żebyś w ten właśnie sposób wpisał każdą z poniższych linii kodu Pythona w interpreterze `python3.6` i zobaczył, co ona robi.

Ćwiczenie 25. — sesja

```
1  import ex25
2  sentence = "To, co najlepsze, spotyka tych, którzy potrafią czekać."
3  words = ex25.break_words(sentence)
4  words
5  sorted_words = ex25.sort_words(words)
6  sorted_words
7  ex25.print_first_word(words)
8  ex25.print_last_word(words)
9  words
10 ex25.print_first_word(sorted_words)
11 ex25.print_last_word(sorted_words)
12 sorted_words
13 sorted_words = ex25.sort_sentence(sentence)
14 sorted_words
15 ex25.print_first_and_last(sentence)
16 ex25.print_first_and_last_sorted(sentence)
```

Kiedy pracuję z modulem `ex25.py` w `python3.6`, wygląda to następująco.

Ćwiczenie 25. — sesja

```
Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import ex25
Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import ex25
>>> sentence = "To, co najlepsze, spotyka tych, którzy potrafią czekać."
>>> words = ex25.break_words(sentence)
>>> words
['To', 'co', 'najlepsze,', 'spotyka', 'tych,', 'którzy', 'potrafią', 'czekać.']
```

```
>>> sorted_words = ex25.sort_words(words)
>>> sorted_words
['To', 'co', 'czekać.', 'którzy', 'najlepsze,', 'potrafią', 'spotyka', 'tych,']
>>> ex25.print_first_word(words)
To
>>> ex25.print_last_word(words)
czekać.
>>> words
['co', 'najlepsze,', 'spotyka', 'tych,', 'którzy', 'potrafią']
>>> ex25.print_first_word(sorted_words)
To
>>> ex25.print_last_word(sorted_words)
tych,
>>> sorted_words
['co', 'czekać.', 'którzy', 'najlepsze,', 'potrafią', 'spotyka']
>>> sorted_words = ex25.sort_sentence(sentence)
>>> sorted_words
['To', 'co', 'czekać.', 'którzy', 'najlepsze,', 'potrafią', 'spotyka', 'tych,']
>>> ex25.print_first_and_last(sentence)
To
czekać.
>>> ex25.print_first_and_last_sorted(sentence)
To
tych,
```

Wpisując każdą z tych linii kodu, upewnij się, że potrafisz odnaleźć uruchamianą funkcję w pliku `ex25.py` i zrozumieć, jak działa. Jeśli otrzymasz inne wyniki lub błąd, będziesz musiał naprawić swój kod, wyjść z interpretera `python3.6` i zacząć od nowa.

Zrób to sam

1. Przeanalizuj pozostałe linie z danych wyjściowych z podrozdziału „Co powinienś zobaczyć” i sprawdź, co robią. Upewnij się, że rozumiesz, w jaki sposób uruchamiasz swoje funkcje w module `ex25`.
2. Spróbuj wpisać polecenia `help(ex25)` oraz `help(ex25.break_words)`. Zwróć uwagę, w jaki sposób otrzymujesz pomoc dla Twojego modułu i jak wyświetlane są te dziwne łańcuchy znaków `"""`, które umieściłeś po każdej funkcji w `ex25`. Te specjalne łańcuchy znaków są nazywane **komentarzami dokumentującymi** i będziesz widywał je częściej.
3. Ciągłe wpisywanie `ex25.` jest denerwujące. Można to skrócić poprzez wykonanie importu w ten sposób: `from ex25 import *`. To tak, jakby powiedzieć: „Zaimportuj wszystko z `ex25`”. Programiści lubią zaczynać od końca. Rozpocznij nową sesję i zobacz, że wszystkie Twoje funkcje tam będą.
4. Spróbuj popsuć swój plik i zobacz, jak będzie wyglądać używanie go w interpreterze `python`. Aby można było go ponownie załadować, będziesz musiał zamknąć Pythona za pomocą polecenia `quit()`.

Typowe pytania

Dla niektórych funkcji otrzymuję `None`. Prawdopodobnie masz funkcję, której brakuje na końcu słowa `return`. Przeczytaj plik wstecz i sprawdź, czy wszystkie linie są poprawne.

Gdy wpisuję `import ex25`, otrzymuję `-bash: import: command not found`. Zwróć uwagę na to, co robię w podrozdziale „Co powinienś zobaczyć”. Robię to w *Pythonie*, a nie w terminalu. Oznacza to, że najpierw uruchamiasz Pythona.

Gdy wpisuję `import ex25.py`, otrzymuję `ImportError: No module named ex25.py`. Nie dodawaj `.py` na końcu. Python „wie”, że plik kończy się na `.py`, więc po prostu wpisz `import ex25`.

Gdy to uruchamiam, otrzymuję `SyntaxError: invalid syntax`. Oznacza to, że brakuje znaku `(` lub `"` albo masz jakiś podobny błąd składniowy w tej linii lub nad nią. Za każdym razem, gdy pojawi się ten błąd, zacznij we wskazanej linii i sprawdź, czy jest prawidłowa, a następnie cofaj się, sprawdzając po kolei każdą linię powyżej.

W jaki sposób funkcja `words.pop` może zmieniać zmienną `words`? To skomplikowane pytanie, ale w tym przypadku `words` jest listą i dzięki temu możesz wykonywać na niej polecenia, a ona zachowa wyniki tych poleceń. Jest to podobne do działania plików i wielu innych rzeczy, gdy wykonujesz na nich jakieś operacje.

Kiedy powinienem używać w funkcji `print` zamiast `return`? Zwracanie z funkcji za pomocą `return` przekazuje wynik do tej linii kodu, która wywołała funkcję. Możesz potraktować funkcję jak coś, co przyjmuje dane wejściowe poprzez swoje argumenty i zwraca dane wyjściowe poprzez `return`. Polecenie `print` jest *zupełnie* z tym niezwiązane i zajmuje się tylko drukowaniem danych wyjściowych w terminalu.

Gratulacje, rozwiąż test!

Już prawie skończyłeś pierwszą część książki. W drugiej części zaczniesz się robić ciekawie. Nauczysz się logiki i będziesz w stanie robić różne użyteczne rzeczy, takie jak dokonywanie wyborów.

Zanim przejdiesz dalej, mam dla Ciebie quiz. Ten quiz będzie *bardzo trudny*, ponieważ wymaga naprawienia kodu napisanego przez kogoś innego. Kiedy jesteś programistą, często masz do czynienia z kodami innych programistów — a także z ich arogancją. Programiści bardzo często twierdzą, że ich kod jest doskonały.

Tacy programiści to głupcy, którzy niezbyt przejmują się innymi. Dobry programista, tak jak dobry naukowiec, zakłada, iż zawsze istnieje *jakieś* prawdopodobieństwo, że jego kod jest błędny. Dobrzy programiści zaczynają od przesłanki, że to ich oprogramowanie jest zepsute, a następnie podejmują działania, aby wykluczyć wszystkie możliwe obszary błędu. Dopiero wtedy mogą pokusić się o stwierdzenie, że prawdopodobnie winien jest kod napisany przez kogoś innego.

W tym ćwiczeniu będziesz musiał poradzić sobie ze złym programistą i naprawić jego kod. Tym kodem będzie skopiowana przeze mnie do pojedynczego pliku treść kilku wybranych ćwiczeń, w której usunąłem losowe znaki i dodałem błędy. Większość błędów podpowie Ci Python, natomiast część z nich to błędy matematyczne, które sam musisz znaleźć. Inne to błędy formatowania lub błędy ortograficzne w łańcuchach znaków.

Wszystkie te błędy są bardzo częstymi pomyłkami popełnianymi przez każdego programistę, nawet doświadczonego.

Twoim zadaniem w tym ćwiczeniu jest poprawienie pliku. Wykorzystaj wszystkie swoje umiejętności, aby go ulepszyć. Najpierw przeanalizuj kod, być może wydrukuj, aby wyedytować go tak, jak zrobiłbyś na przykład z pracą semestralną. Usuń każdy znaleziony błąd. Uruchamiaj skrypt i naprawiaj go, dopóki nie będzie działał idealnie. Staraj się nie prosić nikogo o pomoc. Jeśli utkniesz w jakimś miejscu, zrób sobie przerwę i wróć do tego później.

Jeśli nawet zajmie to wiele dni, rozwiąż ten test.

Celem tego ćwiczenia nie jest wpisywanie kodu, ale naprawienie istniejącego pliku. Plik możesz pobrać z serwera FTP wydawnictwa Helion (<ftp://ftp.helion.pl/przyklady/pyt3pw.zip>). Rozpakuj pobrane archiwum i otwórz plik `ex26.txt`. Skopiuj i wklej kod do pliku o nazwie `ex26.py`. To jest jedyny przypadek, w którym pozwalam Ci kopiować i wklejać.

Typowe pytania

Czy muszę zaimportować `ex25.py`, czy mogę po prostu usunąć odniesienia do niego? Możesz zrobić jedno albo drugie. Ten plik ma jednak funkcje z `ex25`, więc zacznij od usunięcia referencji do niego.

Czy mogę uruchamiać kod, gdy go naprawiam? Jak najbardziej. Komputer ma Ci pomóc, więc używaj go tak często, jak to możliwe.

Zapamiętywanie logiki

Nadszedł dzień, w którym zaczynasz uczyć się logiki. Do tej pory robiłeś wszystko to, co jest możliwe podczas odczytywania oraz zapisywania plików w terminalu i dowiedziałeś się całkiem sporo na temat możliwości wykonywania działań arytmetycznych w Pythonie.

Od teraz będziesz poznawać *logikę*. Nie nauczysz się skomplikowanych teorii badanych na uniwersytetach, ale podstawowej logiki, dzięki której działają rzeczywiste programy i której prawdziwi programiści potrzebują każdego dnia.

Nauka logiki będzie poprzedzona ćwiczeniem pamięciowym. Chcę, żebyś wykonywał to ćwiczenie przez cały tydzień. Nie poddawaj się. Jeśli nawet będziesz potwornie znudzony, rób to dalej. To ćwiczenie obejmuje zestaw tablic logicznych, które musisz zapamiętać, aby ułatwić sobie wykonywanie późniejszych ćwiczeń.

Ostrzegam, że na początku nie będzie to zabawne. Będzie nudne i nużące, ale nauczy Cię bardzo ważnej umiejętności, której będziesz potrzebował jako programista. Będziesz musiał zapamiętywać różne ważne koncepcje w Twoim życiu. Większość z tych koncepcji będzie ekscytyująca, gdy już je opanujesz. Będziesz się z nimi zmagać jak w zapasach z kałamarnicą, ale pewnego dnia je zrozumiesz. Cały wysiłek włożony w zapamiętywanie podstaw odplaci się później z nawiązką.

Oto wskazówka, jak coś zapamiętać i nie oszaleć. Zapamiętuj małe fragmenty na raz przez cały dzień i zaznaczaj to, nad czym musisz najbardziej popracować. Nie próbuj zapamiętać tych tablic za jednym podejściem, siedząc nad nimi przez kilka godzin. To nie zadziała. Twój mózg i tak zachowa tylko to, czego nauczyłeś się przez pierwsze 15 lub 30 minut. Zamiast tego dla każdej pozycji z tablicy przygotuj sobie fiszkę z lewą kolumną (na przykład `True or False`) na jednej stronie karteczki i prawą kolumną na odwrocie. Wyjmuj po kolei fiszki i patrząc na pierwszą stronę, próbuj odgadnąć, co jest na odwrocie. Jeśli wylosujesz na przykład `True or False`, powinieneś natychmiast powiedzieć `True`! Ćwicz tak długo, aż wszystko zapamiętasz.

Kiedy już to zrobisz, zacznij spisywać co wieczór w notatniku swoje własne tablice prawdy. Nie kopiuje ich. Spróbuj robić to z pamięci. Kiedy na czymś utkniesz, szybko zerknij na przygotowane przeze mnie tablice, aby odświeżyć sobie pamięć. W ten sposób wytrenujesz swój umysł i zapamiętasz wszystkie tablice.

Nie poświęcaj na to ćwiczenie więcej niż tydzień, ponieważ będziesz stosować te rzeczy na bieżąco.

Wyrażenie logiczne

W Pythonie stosuje się wymienione poniżej operatory logiczne określające, czy coś jest „prawdziwe”, czy „fałszywe”. Logika na komputerze polega na sprawdzeniu, czy jakaś kombinacja tych operatorów i pewnych zmiennych ma wartość True w danym punkcie programu. Oto operatory logiczne:

- `and` (koniunkcja),
- `or` (alternatywa),
- `not` (negacja),
- `!=` (nie równa się),
- `==` (równa się),
- `>=` (jest większe lub równe),
- `<=` (jest mniejsze lub równe),
- `True` (prawda),
- `False` (fałsz).

Właściwie na większość z nich natknąłeś się już wcześniej, ale na `and`, `or` i `not` prawdopodobnie nie. Faktycznie działają tak, jak mógłbyś się spodziewać, biorąc pod uwagę ich znaczenie w języku mówionym.

Tablice prawdy

Użyjemy teraz tych znaków, aby utworzyć tablice prawdy, które musisz zapamiętać.

NOT	True?
<code>not False</code>	True
<code>not True</code>	False

OR	True?
<code>True or False</code>	True
<code>True or True</code>	True
<code>False or True</code>	True
<code>False or False</code>	False

AND	True?
<code>True and False</code>	False
<code>True and True</code>	True
<code>False and True</code>	False
<code>False and False</code>	False

NOT OR	True?
<code>not(True or False)</code>	False
<code>not(True or True)</code>	False
<code>not(False or True)</code>	False
<code>not(False or False)</code>	True

NOT AND	True?
<code>not(True and False)</code>	True
<code>not(True and True)</code>	False
<code>not(False and True)</code>	True
<code>not(False and False)</code>	True

!=	True?
<code>1 != 0</code>	True
<code>1 != 1</code>	False
<code>0 != 1</code>	True
<code>0 != 0</code>	False

==	True?
<code>1 == 0</code>	False
<code>1 == 1</code>	True
<code>0 == 1</code>	False
<code>0 == 0</code>	True

Teraz na podstawie tych tablic sporządź własne fiszki i poświęć tydzień na ich zapamiętywanie. Pamiętaj jednak, że podczas lektury tej książki nie ma czegoś takiego jak porażka — po prostu staraj się każdego dnia z całych sił, a potem jeszcze *trochę więcej*.

Typowe pytania

Czy nie mogę po prostu nauczyć się zasad algebry Boole’a i nie zapamiętywać tego? Jasne, możesz to zrobić, ale wtedy będziesz musiał ciągle odwoływać się do zasad algebry Boole’a podczas pisania kodu. Jeśli najpierw zapamiętasz te tablice, nie tylko rozwiniesz swoje umiejętności zapamiętywania, ale także spowodujesz, że te operacje staną się dla Ciebie naturalne. Potem koncepcja algebry Boole’a będzie już łatwa, ale zrób tak, jak wolisz.

Ćwiczmy logikę boolowską

Kombinacje logiczne, których nauczyłeś się w ostatnim ćwiczeniu, nazywają się wyrażeniami **logiki boolowskiej**. Logika boolowska jest w programowaniu używana *na każdym kroku*. Jest to podstawowa część obliczeń, a znajomość tych wyrażień można porównać do znajomości skal w muzyce.

W tym ćwiczeniu zaczniesz wykorzystywać w Pythonie logiczne tablice prawdy, których nauczyłeś się ostatnio na pamięć. Zapoznaj się z poniższymi problemami logicznymi i spróbuj zapisać Twoją odpowiedź dla każdego z nich. Możliwe odpowiedzi to `True` lub `False`. Następnie uruchom Pythona w terminalu i wpisz każdy problem logiczny w celu potwierdzenia Twoich odpowiedzi.

1. `True and True`,
2. `False and True`,
3. `1 == 1 and 2 == 1`,
4. `"test" == "test"`,
5. `1 == 1 or 2 != 1`,
6. `True and 1 == 1`,
7. `False and 0 != 0`,
8. `True or 1 == 1`,
9. `"test" == "testowanie"`,
10. `1 != 0 and 2 == 1`,
11. `"test" != "testowanie"`,
12. `"test" == 1`,
13. `not (True and False)`,
14. `not (1 == 1 and 0 != 1)`,
15. `not (10 == 1 or 1000 == 1000)`,
16. `not (1 != 10 or 3 == 4)`,
17. `not ("testowanie" == "testowanie" and "Zed" == "Fajny Gość")`,
18. `1 == 1 and (not ("testowanie" == 1 or 1 == 0))`,
19. `"tłusty" == "boczek" and (not (3 == 4 or 3 == 3))`,
20. `3 == 3 and (not ("testowanie" == "testowanie" or "Python" == "Zabawa"))`.

Pod koniec ćwiczenia pokażę również sztuczkę, która pomoże Ci rozwiązywać bardziej skomplikowane problemy.

Za każdym razem, gdy zobaczysz podobne zdanie logiczne, możesz je łatwo rozwiązać, wykonując poniższe proste kroki.

1. Znajdź test równości (`==` lub `!=`) i zastąp go wynikiem.
2. Znajdź wszystkie `and` oraz `or` w nawiasach i rozwiąż je w pierwszej kolejności.
3. Znajdź każde `not` i odwróć je.
4. Znajdź pozostałe `and` oraz `or` i rozwiąż je.
5. Kiedy skończysz, powinieneś otrzymać `True` lub `False`.

Zademonstruję to na podstawie wariacji na temat problemu 20.:

```
3 != 4 and not ("testowanie" != "test" or "Python" == "Python")
```

Wykonam każdy z powyższych kroków, dopóki nie sprowadzę wszystkiego do pojedynczego wyniku.

1. Najpierw rozwiązuję każdy test równości:

```
3 != 4 to True: True and not ("testowanie" != "test" or "Python" == "Python")
"testowanie" != "test" to True: True and not (True or "Python" == "Python")
"Python" == "Python" to True: True and not (True or True)
```

2. Potem znajduję każde `and` i `or` w nawiasach:

```
(True or True) to True: True and not (True)
```

3. Następnie znajduję każde `not` i odwracam je:

```
not (True) to False: True and False
```

4. Na koniec znajduję pozostałe `and` oraz `or` i rozwiązuję je:

```
True and False to False
```

Na tym kończę i wiem, że wynik to `False`.

OSTRZEŻENIE! Bardziej skomplikowane zdania logiczne mogą z początku wydawać się *bardzo trudne*. Powinieneś mieć wystarczającą wiedzę, żeby spróbować je rozwiązywać, ale jeśli nie będzie Ci to wychodzić, nie zniechęcaj się. Przygotowuję Cię po prostu na więcej takiej „gimnastyki logicznej”, aby później te fajne rzeczy były znacznie łatwiejsze. Po prostu rób dalej to, co robisz, i notuj, co robisz źle, ale nie przejmuj się, że nie wchodzi Ci to jeszcze całkowicie do głowy. Wszystko przyjdzie z czasem.

Co powinieneś zobaczyć

Gdy spróbujesz już zgadnąć odpowiedzi na zadane problemy, możesz uruchomić sesję z Pythonem, która będzie wyglądać tak.

```
$ python3.6
Python 2.5.1 (r251:54863, Feb 6 2009, 19:02:12)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> True and True
True
>>> 1 == 1 and 2 == 2
True
```

Zrób to sam

1. W Pythonie jest wiele operatorów podobnych do `!=` i `==`. Postaraj się znaleźć możliwe dużo „operatorów porównania”. Mogą to być na przykład operatory `<` lub `<=`.
2. Napisz nazwy dla każdego z tych operatorów porównania. Ja na przykład dla operatora `!=` używam nazwy „nie równa się”.
3. Pobaw się z Pythonem, wpisując nowe operatory logiczne, ale zanim naciśniesz *Enter*, spróbuj odgadnąć wynik. Nie myśl o tym. Wykrzyknij pierwszą rzecz, która przyjdzie Ci na myśl. Zapisz, naciśnij *Enter* i śledź, ile będziesz miał dobrych i złych odpowiedzi.
4. Gdy skończysz, wyrzuć kartkę papieru używaną w punkcie 3., żebyś przypadkowo nie próbował jej użyć później.

Typowe pytania

Dlaczego "test" and "test" zwraca "test" lub 1 and 1 zwraca 1 zamiast True?

Python, tak jak wiele innych języków, lubi zwracać raczej jeden z operandów wyrażeń boolowskich zamiast po prostu `True` lub `False`. Oznacza to, że jeśli wpiszesz `False and 1`, otrzymasz pierwszy operand (`False`), a jeśli wpiszesz `True and 1`, otrzymasz drugi (1). Pobaw się tym trochę.

Czy jest jakaś różnica między `!=` i `<>`? Python zarzucił używanie `<>` na korzyść `!=`, więc użyj operatora `!=`. Poza tym nie powinno być żadnej różnicy.

Czy jest jakiś skrót? Tak. Każde wyrażenie `and` z operandem `False` daje natychmiast `False`, a każde wyrażenie `or` zawierające `True` daje od razu `True`. Upewnij się jednak, że potrafisz przetworzyć całe wyrażenie, ponieważ później będzie to pomocne.

Co, jeśli...

O to kolejny skrypt Pythona, który wprowadza instrukcję `if`. Wpisz poniższy kod i upewnij się, że działa prawidłowo, a przekonasz się, że Twoje ćwiczenie się opłaciło.

ex29.py

```
1  people = 20
2  cats = 30
3  dogs = 15
4
5
6  if people < cats:
7      print("Zbyt dużo kotów! Świat jest skazany na zagładę!")
8
9  if people > cats:
10     print("Nie za dużo kotów! Świat jest ocalony!")
11
12 if people < dogs:
13     print("Świat się ślini!")
14
15 if people > dogs:
16     print("Świat jest suchy!")
17
18
19 dogs += 5
20
21 if people >= dogs:
22     print("Liczba ludzi jest większa lub równa liczbie psów.")
23
24 if people <= dogs:
25     print("Liczba ludzi jest mniejsza lub równa liczbie psów.")
26
27 if people == dogs:
28     print("Ludzie są psami.")
```

Co powinieneś zobaczyć

Ćwiczenie 29. — sesja

```
$ python3.6 ex29.py
Zbyt dużo kotów! Świat jest skazany na zagładę!
Świat jest suchy!
Liczba ludzi jest większa lub równa liczbie psów.
Liczba ludzi jest mniejsza lub równa liczbie psów.
Ludzie są psami.
```

Zrób to sam

W tym podrozdziale „Zrób to sam” postaraj się odgadnąć, czym — według Ciebie — jest instrukcja `if` i co robi. Zanim przejdziesz do następnego ćwiczenia, spróbuj własnymi słowami odpowiedzieć na poniższe pytania.

1. Jak myślisz, co robi instrukcja `if` z kodem znajdującym się pod nią?
2. Dlaczego kod pod instrukcją `if` musi być wcięty o cztery spacje?
3. Co się dzieje, jeśli kod nie jest wcięty?
4. Czy możesz wstawić w instrukcji `if` inne wyrażenia boolowskie z ćwiczenia 27.? Spróbuj.
5. Co się stanie, jeśli zmienisz początkowe wartości dla zmiennych `people` (ludzie), `cats` (koty) i `dogs` (psy)?

Typowe pytania

Co znaczy `+=`? Kod `x += 1` oznacza to samo, co `x = x + 1`, ale wymaga mniej wpisywania. Możesz to nazwać operatorem inkrementacji. To samo dotyczy `-=` i wielu innych wyrażań, które poznasz później.

Else oraz if

W poprzednim ćwiczeniu użyłeś kilku instrukcji `if`, a następnie próbowałeś odgadnąć, czym są i jak działają. Zanim nauczysz się więcej, wyjaśnię wszystko, podając odpowiedzi na pytania z podrozdziału „Zrób to sam” ostatniego ćwiczenia. Spróbowałeś samodzielnie odpowiedzieć na te pytania, prawda?

1. Jak myślisz, co robi instrukcja `if` z kodem znajdującym się pod nią? Instrukcja `if` tworzy w kodzie tak zwaną „gałąź”. Przypomina to przygodowe książki gry, w których po dokonaniu pewnego wyboru jesteś kierowany na określoną stronę, a gdy wybierasz inaczej, idziesz w innym kierunku. Instrukcja `if` mówi skryptowi: „Jeśli to wyrażenie logiczne ma wartość `True`, uruchom znajdujący się poniżej kod; w przeciwnym razie pomiń go”.
2. Dlaczego kod pod instrukcją `if` musi być wcięty o cztery spacje? Dwukropek na końcu linii informuje Pythona, że zamierzasz utworzyć nowy „blok” kodu, a cztery spacje wcięcia wskazują, które linie kodu znajdują się w tym bloku. Jest to *dokładnie* to samo, co robiłeś przy tworzeniu funkcji w pierwszej części książki.
3. Co się dzieje, jeśli kod nie jest wcięty? Jeśli kod nie jest wcięty, najprawdopodobniej otrzymasz błąd Pythona. Python oczekuje, że *cokolwiek* wpiszesz po zakończeniu linii dwukropkiem (:), będzie wcięte.
4. Czy możesz wstawić w instrukcji `if` inne wyrażenia boolowskie z ćwiczenia 27.? Spróbuj. Tak, możesz i mogą być one dowolnie skomplikowane, chociaż ogólnie rzecz biorąc, naprawdę skomplikowane rzeczy to zły styl kodowania.
5. Co się stanie, jeśli zmienisz początkowe wartości dla zmiennych `people` (ludzie), `cats` (koty) i `dogs` (psy)? Ponieważ porównujesz liczby, jeśli je zmienisz, inne instrukcje `if` będą ewaluowały do `True` i uruchomione zostaną znajdujące się pod nimi bloki kodu. Wróć do tego ćwiczenia, podstaw inne liczby i sprawdź, czy potrafisz odgadnąć, które bloki kodu zostaną uruchomione.

Porównaj moje odpowiedzi ze swoimi i upewnij się, że *naprawdę* rozumiesz koncepcję „bloku” kodu. Będzie to ważne w następnym ćwiczeniu, gdzie napiszesz wszystkie części instrukcji `if`, których możesz użyć.

Wpisz poniższy kod i upewnij się, że działa prawidłowo.

ex30.py

```

1  people = 30
2  cars = 40
3  trucks = 15
4
5
6  if cars > people:
7      print("Powinniśmy jechać samochodami.")
8  elif cars < people:
```



```
9     print("Nie powinniśmy jechać samochodami.")
10 else:
11     print("Nie możemy się zdecydować.")
12
13 if trucks > cars:
14     print("Jest zbyt dużo ciężarówek.")
15 elif trucks < cars:
16     print("Może powinniśmy wziąć ciężarówkę.")
17 else:
18     print("Nadal nie możemy się zdecydować.")
19
20 if people > trucks:
21     print("W porządku, po prostu weźmy ciężarówkę.")
22 else:
23     print("Dobra, w takim razie zostajemy w domu.")
```

Co powinieneś zobaczyć

Ćwiczenie 30. — sesja

```
$ python3.6 ex30.py
Powinniśmy jechać samochodami.
Może powinniśmy wziąć ciężarówkę.
W porządku, po prostu weźmy ciężarówkę.
```

Zrób to sam

1. Spróbuj zgadnąć, co robią instrukcje `elif` i `else`.
2. Zmień liczbę samochodów (`cars`), osób (`people`) i ciężarówek (`trucks`), a następnie prześledź każdą instrukcję `if`, aby zobaczyć, co zostanie wydrukowane.
3. Wypróbuj bardziej złożone wyrażenia logiczne, na przykład `cars > people` or `trucks < cars`.
4. Nad każdą linią kodu napisz komentarz opisujący, co robi.

Typowe pytania

Co się stanie, jeśli wiele bloków `elif` będzie miało wartość `True`? Python zaczyna od góry i uruchamia pierwszy blok o wartości `True`, więc uruchomi tylko jeden blok.

Podjmowanie decyzji

W pierwszej części książki głównie drukowałeś łańcuchy znaków i wywoływałeś funkcje, ale wszystko szło zasadniczo po linii prostej. Skrypty były wykonywane od góry do dołu, gdzie się kończyły. Jeśli utworzyłeś funkcję, mogłeś uruchomić ją później, ale nadal nie miała ona tego rodzaju rozgałęzienia, które jest potrzebne, żeby naprawdę podejmować decyzję. Ponieważ teraz masz do dyspozycji instrukcje `if`, `else` i `elif`, możesz zacząć pisać skrypty dokonujące wyborów.

W poprzednim skrypcie napisałeś prosty zestaw testów zadających pewne pytania. W tym skrypcie będziesz zadawał użytkownikowi pytania i podejmował decyzje na podstawie jego odpowiedzi. Wpisz ten skrypt, a następnie pobaw się nim na tyle długo, aby go rozgryźć.

ex31.py

```
1  print("""Wchodzisz do ciemnego pokoju z dwoma drzwiami.  
2  Przechodzisz przez drzwi nr 1 czy nr 2?""")  
3  
4  door = input("> ")  
5  
6  if door == "1":  
7      print("Widzisz tam wielkiego niedźwiedzia, który zajada sernik.")  
8      print("Co robisz?")  
9      print("1. Zabierasz sernik.")  
10     print("2. Krzyczysz na niedźwiedzia.")  
11  
12     bear = input("> ")  
13  
14     if bear == "1":  
15         print("Niedźwiedź odgryza Ci nos. Dobra robota!")  
16     elif bear == "2":  
17         print("Niedźwiedź odgryza Ci nogi. Dobra robota!")  
18     else:  
19         print(f"Cóż, {bear} to prawdopodobnie lepszy wybór.")  
20         print("Niedźwiedź ucieka.")  
21  
22 elif door == "2":  
23     print("Wpatrujesz się w nieskończoną otchłń oka Cthulhu.")  
24     print("1. Jagody.")  
25     print("2. Żółte spinacze do bielizny.")  
26     print("3. Wyzrozumiałe rewolwery nucią melodie.")  
27  
28     insanity = input("> ")  
29  
30     if insanity == "1" or insanity == "2":  
31         print("Twoje ciało ocalało, ale masz mózg jak galaretka owocowa.")  
32         print("Dobra robota!")  
33     else:
```

```
34         print("Z szaleństwa gniją Ci oczy i zamieniają się w kałużę błota.")
35         print("Dobra robota!")
36
37     else:
38         print("Potykasz się, nadziewasz na nóż i umierasz. Dobra robota!")
```

Kluczową sprawą jest tutaj to, że teraz umieszczasz instrukcje *if* *wewnątrz instrukcji if* jako kod, który może zostać uruchomiony. Jest to bardzo użyteczne narzędzie i może być wykorzystywane do tworzenia „zagnieżdżonych” decyzji, w których jedna gałąź prowadzi do kolejnej i kolejnej, i tak dalej.

Upewnij się, że rozumiesz koncepcję instrukcji *if* *wewnątrz instrukcji if*. Aby to dobrze pojąć, wykonaj ćwiczenia z podrozdziału „Zrób to sam”.

Co powinieneś zobaczyć

Tutaj zagrałem w tę małą przygodówkę. Nie poszło mi za dobrze.

Ćwiczenie 31. — sesja

```
$ python3.6 ex31.py
Wchodzisz do ciemnego pokoju z dwoma drzwiami.
Przechodzisz przez drzwi nr 1 czy nr 2?
> 1
Widzisz tam wielkiego niedźwiedzia, który zajada sernik.
Co robisz?
1. Zabierasz sernik.
2. Krzyczysz na niedźwiedzia.
> 2
Niedźwiedź odgryza Ci nogi. Dobra robota!
```

Zrób to sam

1. Dopisz nowe części gry i pozmieniał decyzje, które może podejmować użytkownik. Rozwijaj tę grę, na ile się da, zanim zrobi się niedorzeczna.
2. Napisz zupełnie nową grę. Może ta Ci się nie podoba, więc opracuj własną. W końcu to Twój komputer i możesz robić, co chcesz.

Typowe pytania

Czy mogę zastąpić *elif* sekwencją kombinacji *if-else*? W niektórych sytuacjach możesz, ale to zależy od tego, jak napisane jest każde *if/else*. Oznacza to również, że Python sprawdzi *każdą* kombinację *if-else*, a nie tylko pierwsze fałszywe, tak jak w przypadku *if-elif-else*. Wypróbuj różne możliwości, aby poznać różnice.

Jak określić, czy liczba mieści się w pewnym przedziale liczbowym? Masz dwie opcje. Użyj $0 < x < 10$ lub $1 \leq x < 10$, co jest notacją klasyczną albo zastosuj `x in range(1, 10)`.

A jeśli chciałbym więcej opcji w blokach if-elif-else? Dodaj więcej bloków `elif` dla każdego możliwego wyboru.

Pętle i listy

Powinieneś już poradzić sobie z pisaniem o wiele bardziej interesujących programów. Jeśli byłeś pilnym uczniem, zapewne już wiesz, że teraz możesz połączyć wszystkie te rzeczy, których nauczyłeś się wcześniej, z instrukcjami `if` i wyrażeniami boolowskimi, aby Twoje programy robiły różne inteligentne rzeczy.

Jednak programy muszą również bardzo szybko wykonywać pewne powtarzalne czynności. W tym ćwiczeniu użyjemy pętli `for` do budowania i drukowania różnych list. Kiedy wykonasz ćwiczenie, zaczniesz rozumieć, czym są pętle. Nie powiem Ci tego teraz. Musisz rozgryźć to sam.

Zanim będziesz mógł użyć pętli `for`, potrzebujesz sposobu *przechowywania* wyników wykonywania tej pętli. Do tego celu najlepiej wykorzystać listy. Lista jest dokładnie tym, na co wskazuje nazwa: kontenerem elementów, które są uporządkowane w kolejności od pierwszego do ostatniego. Nie jest to skomplikowane; musisz tylko nauczyć się nowej składni. Najpierw pokażę, jak tworzyć listy.

```

hairs = ['brązowe', 'blond', 'rude']
eyes = ['brązowe', 'niebieskie', 'zielone']
weights = [1, 2, 3, 4]

```

Listę rozpoczynasz od lewego nawiasu kwadratowego (`[`), który ją „otwiera”. Następnie umieszczasz w liście wszystkie żądane elementy, rozdzielając je przecinkami, podobnie jak argumenty funkcji. Listę zamykasz prawym nawiasem kwadratowym (`]`), który oznacza jej zakończenie. Następnie Python pobiera tę listę z całą jej zawartością i przypisuje ją do zmiennej.

OSTRZEŻENIE! Dla tych, którzy nie potrafią kodować, tutaj sprawy się komplikują. Twój mózg nauczył się, że świat jest płaski. Pamiętasz, jak w ostatnim ćwiczeniu umieszczałeś instrukcje `if` wewnątrz instrukcji `if`? Prawdopodobnie rozboleła Cię od tego głowa, ponieważ większość ludzi nie zastanawia się, jak „zagnieżdżać” jedno rzeczy wewnątrz drugich. W programowaniu struktury zagnieżdżone są wszędzie. Znajdziesz tam funkcje, które wywołują inne funkcje, które zawierają instrukcje `if`, które mają listy z listami wewnątrz list. Jeśli zobaczysz taką strukturę, której nie możesz rozgryźć, wyciągnij ołówek, papier i przeanalizuj ją ręcznie krok po kroku, aż wszystko zrozumiesz.

Teraz za pomocą pętli `for` zbudujemy kilka list i wydrukujemy je.

ex32.py

```

1  the_count = [1, 2, 3, 4, 5]
2  fruits = ['jabłko', 'pomarańcze', 'gruszki', 'morele']
3  change = [1, 'jednogroszówki', 2, 'dwugroszówki', 3, 'pięciogroszówki']

```

```
4
5 # to jest pierwszy rodzaj pętli for, która przechodzi przez listę
6 for number in the_count:
7     print(f"To jest liczba {number}")
8
9 # to samo, co wyżej
10 for fruit in fruits:
11     print(f"Rodzaj owocu: {fruit}")
12
13 # możemy również przechodzić przez listy mieszane
14 # zwróć uwagę, że musimy użyć {}, ponieważ nie wiemy, co w niej jest
15 for i in change:
16     print(f"Mam {i}")
17
18 # możemy również budować listy; zaczniemy od pustej
19 elements = []
20
21 # następnie używamy funkcji range, aby odliczyć od 0 do 5
21 for i in range(0, 6):
23     print(f"Dodajemy {i} do tej listy.")
24     # append jest funkcją, którą listy rozumieją
25     elements.append(i)
26
27 # teraz możemy je również wydrukować
28 for i in elements:
29     print(f"Tym elementem było: {i}")
```

Co powinieneś zobaczyć

Ćwiczenie 32. — sesja

```
$ python3.6 ex32.py
To jest liczba 1
To jest liczba 2
To jest liczba 3
To jest liczba 4
To jest liczba 5
Rodzaj owocu: jabłko
Rodzaj owocu: pomarańcze
Rodzaj owocu: gruszki
Rodzaj owocu: morele
Mam 1
Mam jednogroszówki
Mam 2
Mam dwugroszówki
Mam 3
Mam pięciogroszówki
Dodajemy 0 do tej listy.
Dodajemy 1 do tej listy.
Dodajemy 2 do tej listy.
Dodajemy 3 do tej listy.
Dodajemy 4 do tej listy.
```

```
Dodajemy 5 do tej listy.  
Tym elementem było: 0  
Tym elementem było: 1  
Tym elementem było: 2  
Tym elementem było: 3  
Tym elementem było: 4  
Tym elementem było: 5
```

Zrób to sam

1. Przyjrzyj się wykorzystaniu `range`. Poszukaj w internecie informacji na temat funkcji `range`, aby zrozumieć, jak działa.
2. Czy mogłeś całkowicie uniknąć tej pętli `for` w linii 22. i po prostu przypisać `range(0,6)` bezpośrednio do `elements`?
3. Znajdź dokumentację Pythona na temat list i poczytaj o nich. Jakie inne operacje poza `append` możesz wykonywać na listach?

Typowe pytania

Jak utworzyć listę dwuwymiarową (2D)? Jest to lista w liście, na przykład `[[1,2,3],[4,5,6]]`.

Czy listy i tablice nie są tym samym? To zależy od języka i implementacji. W ujęciu klasycznym lista znacznie różni się od tablicy ze względu na sposób implementacji. Jednak w języku Ruby nazywa się je tablicami. W Pythonie noszą miano list. Na razie nazywaj je po prostu listami, ponieważ właśnie tak określa je Python.

Dlaczego pętla `for` może używać zmiennej, która nie jest jeszcze zdefiniowana? Zmienna jest definiowana przez pętlę `for` po jej uruchomieniu i za każdym razem inicjowana do wartości bieżącego elementu iteracji pętli.

Dlaczego `for i in range(1, 3):` wykonuje pętlę tylko dwa razy, a nie trzy razy? Funkcja `range()` określa przedział prawostronnie otwarty, czyli z *wyłączeniem ostatniej liczby z przedziału*. Dlatego w tym przykładzie zatrzymuje się na drugiej pętli, a nie na trzeciej. Okazuje się, że jest to najczęstszy sposób wykonywania tego rodzaju pętli.

Co robi `elements.append()`? Po prostu dołącza element do końca listy. Otwórz powłokę Pythona i wypróbuj kilka przykładów z listą, którą utworzyłeś. Za każdym razem, gdy napotkasz takie rzeczy, zawsze spróbuj pobawić się nimi interaktywnie w powłoce Pythona.

Pętla while

Teraz całkowicie zaskoczę Cię nową pętlą; oto `while`. Pętla ta wykonuje znajdujący się pod nią blok kodu tak długo, jak długo dane wyrażenie boolowskie ma wartość `True`.

Nadążasz za terminologią, prawda? Jeśli napiszemy linię kodu i zakończymy ją dwukropkiem (:), poinstruujemy Pythona, aby rozpoczął nowy blok kodu, zgadza się? Następnie robimy wcięcie i to jest ten nowy kod. Chodzi tu przede wszystkim o stosowanie w programach odpowiedniej struktury, aby Python „wiedział”, co masz na myśli. Jeśli jeszcze nie zrozumiałeś tej koncepcji, wróć do wcześniejszych ćwiczeń i popracuj trochę z instrukcjami `if`, funkcjami i pętlą `for`, aż w końcu to zrozumiesz.

Później wykonamy kilka ćwiczeń, w których będziesz trenował odczytywanie tych struktur w podobny sposób, jak wbijałem Ci do głowy wyrażenia boolowskie.

Wróćmy do pętli `while`. Wykonują one po prostu test podobny do instrukcji `if`, ale zamiast uruchamiać blok kodu *jeden raz*, przeskakują z powrotem do „góry” do instrukcji `while` i powtarzają pętlę. Pętla `while` jest wykonywana dopóki, dopóty wyrażenie nie będzie miało wartości `False`.

Z pętlami `while` wiąże się pewien problem — czasami nie przestają działać. To świetnie, jeśli masz ochotę po prostu wykonywać pętlę aż do końca wszechświata. W przeciwnym razie prawie zawsze chcesz, aby Twoja pętla kiedyś się zakończyła.

Aby uniknąć tych problemów, musisz przestrzegać następujących zasad.

1. Używaj pętli `while` oszczędnie. Zwykle lepsza jest pętla `for`.
2. Przejrzyj swoje instrukcje `while` i upewnij się, że test logiczny w jakim punkcie osiągnie wartość `False`.
3. Gdy masz wątpliwości, wydrukuj zmienną testową u góry i na dole pętli `while`, aby zobaczyć, co ta pętla robi.

W tym ćwiczeniu nauczysz się korzystać z pętli `while`, wykonując trzy kontrole.

ex33.py

```
1 i = 0
2 numbers = []
3
4 while i < 6:
5     print(f"Na górze i ma wartość {i}")
6     numbers.append(i)
7
8     i = i + 1
9     print("Aktualne liczby: ", numbers)
10    print(f"Na dole i ma wartość {i}")
11
```

```
12
13     print("Te liczby to: ")
14
15     for num in numbers:
16         print(num)
```

Co powinieneś zobaczyć

Ćwiczenie 33. — sesja

```
$ python3.6 ex33.py
Na górze i ma wartość 0
Aktualne liczby: [0]
Na dole i ma wartość 1
Na górze i ma wartość 1
Aktualne liczby: [0, 1]
Na dole i ma wartość 2
Na górze i ma wartość 2
Aktualne liczby: [0, 1, 2]
Na dole i ma wartość 3
Na górze i ma wartość 3
Aktualne liczby: [0, 1, 2, 3]
Na dole i ma wartość 4
Na górze i ma wartość 4
Aktualne liczby: [0, 1, 2, 3, 4]
Na dole i ma wartość 5
Na górze i ma wartość 5
Aktualne liczby: [0, 1, 2, 3, 4, 5]
Na dole i ma wartość 6
Te liczby to:
0
1
2
3
4
5
```

Zrób to sam

1. Przekształć tę pętlę while w funkcję, którą możesz wywoływać, i zastąp 6 w teście ($i < 6$) zmienną.
2. Użyj tej funkcji do przepisania skryptu, aby wypróbować inne liczby.
3. Dodaj do argumentów funkcji kolejną zmienną, którą możesz przekazać, pozwalającą zmienić $+ 1$ w linii 8., żebyś mógł ustalić, o jaką wartość będziesz inkrementował zmienną i .
4. Przepisz skrypt ponownie, aby skorzystać z tej funkcji i zobaczyć, jaki daje efekt.
5. Napisz ten skrypt przy użyciu pętli for i funkcji range. Czy nadal potrzebujesz inkrementora w środku kodu? Co się stanie, kiedy się go nie pozbędiesz?

Jeśli kiedykolwiek podczas wykonywania tego ćwiczenia Twój skrypt zacznie wariować (a prawdopodobnie tak będzie), po prostu przytrzymaj *Ctrl* i naciśnij *C* (*Ctrl*+*C*), a program się zakończy.

Typowe pytania

Jaka jest różnica między pętlami *for* i *while*? Pętla *for* może iterować (przechodzić) tylko „przez” kolekcje elementów. Natomiast pętla *while* może wykonywać dowolnego rodzaju iteracje. Jednak pętle *while* są trudniejsze do prawidłowego napisania, a wiele rzeczy można zazwyczaj zrobić za pomocą pętli *for*.

Pętle są trudne. Jak je rozgryźć? Ludzie nie rozumieją pętli przede wszystkim z tego powodu, że nie mogą nadążyć za „skokami”, które wykonuje kod. Kiedy pętla działa, przechodzi przez swój blok kodu, a na końcu przeskakuje z powrotem na górę. Aby to sobie zwizualizować, umieść w pętli instrukcje *print*, które będą pokazywać, w którym miejscu wykonywania pętli jest Python i jakie są wartości zmiennych w tych punktach. Napisz linie *print* przed pętlą, w górnej części pętli, w środku i na dole. Przestudiuj dane wyjściowe i spróbuj zrozumieć skoki, które wykonuje pętla.

Uzyskiwanie dostępu do elementów list

Listy są całkiem przydatne, ale jeśli nie możesz dostać się do znajdujących się w nich elementów, przestają być użyteczne. Potrafisz już przeglądać elementy listy w kolejności, ale co miałbyś zrobić, aby otrzymać na przykład piąty element? Musisz wiedzieć, jak uzyskiwać dostęp do elementów listy. Oto sposób uzyskania dostępu do *pierwszego* elementu listy:

```
animals = ['niedźwiedź', 'tygrys', 'pingwin', 'zebra']  
bear = animals[0]
```

Bierzesz listę zwierząt (`animals`), a następnie dostajesz pierwsze zwierzę za pomocą `0`?! Jak to działa? Takie są zasady matematyczne i w Pythonie listę rozpoczyna się od `0`, a nie od `1`. Wydaje się to dziwne, ale ma wiele zalet.

Najlepiej wyjaśnię to, pokazując różnicę między sposobem używania liczb przez Ciebie i przez programistów.

Wyobraź sobie, że oglądasz wyścig czterech zwierząt z naszej listy (`['niedźwiedź', 'tygrys', 'pingwin', 'zebra']`). Przekraczają linię mety w *kolejności*, w jakiej znajdują się na tej liście. Wyścig był naprawdę ekscytujący, ponieważ zwierzęta nie pozjadały się nawzajem i jakoś udało się im pobiec w wyścigu. Twój znajomy przychodzi spóźniony i chce wiedzieć, które zwierzę wygrało. Czy zapytałby: „Hej, które przybiegło na miejscu zero?”. Nie, zapyta: „Hej, które przybiegło *pierwsze*?”.

Ważna jest *kolejność* zwierząt. Nie możesz mieć drugiego zwierzęcia bez pierwszego i nie możesz mieć trzeciego bez drugiego. Niemożliwe jest również „zerowe” zwierzę, ponieważ zero oznacza nic. Jak nic może wygrać wyścig? To po prostu nie ma sensu. Takie liczby nazywamy liczbami „porządkowymi”, ponieważ wskazują porządek rzeczy.

Jednak programiści nie mogą myśleć w ten sposób, gdyż muszą mieć możliwość wybrania dowolnego elementu z listy w dowolnym momencie. Dla programistów lista zwierząt przypomina talię kart. Jeśli chcą tygrysa, wyciągają go z talii. Jeśli chcą zebry, również mogą ją wyciągnąć. Ten wymóg wyciągania losowo elementów z listy oznacza, że potrzeba sposobu, aby konsekwentnie wskazywać elementy według adresu lub „indeksu”, a najlepiej rozpoczynać numerację indeksów od `0`. Zaufaj mi: zasady matematyczne *dużo* lepiej nadają się do określania takiego dostępu. Tego rodzaju liczba to liczba „kardynalna”, co oznacza, że możesz wybierać losowo, więc musi być element `0`.

Jak to może Ci pomóc w pracy z listami? To proste. Za każdym razem, gdy powiesz sobie: „Chcę trzecie zwierzę”, przetłumacz tę liczbę porządkową na kardynalną, odejmując `1`. „Trzecie” zwierzę znajduje się pod indeksem `2` i jest pingwinem. Musisz tak robić, ponieważ całe życie posługiwałeś się liczbami porządkowymi, a teraz musisz myśleć w kategoriach liczb kardynalnych. Po prostu odejmij `1` i wszystko będzie w porządku.

Zapamiętaj: porządkowe == uporządkowane, pierwszy; kardynalne == losowe karty, 0.

Poćwiczmy to. Wykorzystaj poniższą listę zwierząt i wykonaj ćwiczenie, w którym będziesz zapisywał zwierzę odpowiadające wskazanej liczbie porządkowej lub kardynalnej. Pamiętaj, jeśli mówię „pierwszy”, „drugi”, to używam liczb porządkowych, więc odejmij 1. Jeśli podam Ci liczbę kardynalną (na przykład „Zwierzę z pozycji 1”), użyj jej bezpośrednio.

```
animals = ['niedźwiedź', 'python3.6', 'paw', 'kangur', 'wieloryb', 'dziobak']
```

1. Zwierzę z pozycji 1.
2. Trzecie zwierzę.
3. Pierwsze zwierzę.
4. Zwierzę z pozycji 3.
5. Piąte zwierzę.
6. Zwierzę z pozycji 2.
7. Szóste zwierzę.
8. Zwierzę z pozycji 4.

Dla każdego punktu zapisz pełne zdanie w postaci: „Pierwsze zwierzę znajduje się pod indeksem 0 i jest to niedźwiedź”. Następnie zapisz to odwrotnie: „Zwierzę o indeksie 0 jest pierwszym zwierzęciem i jest to niedźwiedź”.

Użyj Pythona, aby sprawdzić swoje odpowiedzi.

Zrób to sam

1. Czy znając różnicę pomiędzy tymi typami liczb, możesz wyjaśnić dlaczego rok 2010 w dacie „1 stycznia 2010 r.” to naprawdę rok 2010, a nie 2009? (Podpowiedź: nie możesz wybierać lat losowo).
2. Napisz więcej list i popracuj w podobny sposób z indeksami, aż nauczysz się je swobodnie tłumaczyć.
3. Użyj Pythona, aby sprawdzić swoje odpowiedzi.

OSTRZEŻENIE! Programiści powiedzą Ci, żebyś w tej kwestii poczytał niejakiego *Dijkstrę*. Jednak ja polecam, żebyś unikał jego opracowań w tym temacie, chyba że lubisz, gdy krzyczy na Ciebie ktoś, kto przestał programować w tym samym momencie, w którym programowanie się zaczęło.

Gałęzie i funkcje

Poznałeś instrukcje `if`, funkcje i listy. Teraz nadszedł czas, aby trochę pogimnastykować umysł. Wpisz poniższy kod i spróbuj zrozumieć, co robi.

ex35.py

```

1  from sys import exit
2
3  def gold_room():
4      print("Ten pokój jest pełen złota. Ile złota zabierasz?")
5
6      choice = input("> ")
7      if "0" in choice or "1" in choice:
8          how_much = int(choice)
9      else:
10         dead("Stary, naucz się wpisywać liczby.")
11
12     if how_much < 50:
13         print("Miło, nie jesteś chciwy, wygrywasz!")
14         exit(0)
15     else:
16         dead("Ty chciwy draniu!")
17
18
19  def bear_room():
20      print("Jest tutaj niedźwiedź.")
21      print("Niedźwiedź ma beczkę miodu.")
22      print("Ten gruby niedźwiedź siedzi przed następnymi drzwiami.")
23      print("Jak przesuniesz niedźwiedzia?")
24      bear_moved = False
25
26      while True:
27          choice = input("> ")
28
29          if choice == "zabieram miód":
30              dead("Niedźwiedź spogląda na Ciebie i wali w twarz.")
31          elif choice == "śmieję się z niedźwiedzia" and not bear_moved:
32              print("Niedźwiedź odsunął się od drzwi.")
33              print("Możesz teraz przez nie przejść.")
34              bear_moved = True
35          elif choice == "śmieję się z niedźwiedzia" and bear_moved:
36              dead("Niedźwiedź się wkurzył i odgryzł Ci nogę.")
37          elif choice == "otwieram drzwi" and bear_moved:
38              gold_room()
39          else:
40              print("Nie mam pojęcia, co to znaczy.")
41
42

```



```
43 def cthulhu_room():
44     print("Widzisz wielkiego złego Cthulhu.")
45     print("On, znaczy to, nieważne, wpatruje się w Ciebie, a Ty popadasz w obłąd.")
46     print("Ratujesz się ucieczką, czy zjadasz swoją głowę?")
47
48     choice = input("> ")
49
50     if "ucieczką" in choice:
51         start()
52     elif "głowę" in choice:
53         dead("To było pyszne!")
54     else:
55         cthulhu_room()
56
57
58 def dead(why):
59     print(why, "Dobra robota!")
60     exit(0)
61
62 def start():
63     print("Znajdujesz się w mrocznym pokoju.")
64     print("Po lewej i po prawej znajdują się drzwi.")
65     print("Które wybierasz?")
66
67     choice = input("> ")
68
69     if choice == "lewe":
70         bear_room()
71     elif choice == "prawe":
72         cthulhu_room()
73     else:
74         dead("Błąkasz się po pokoju, aż w końcu umierasz z głodu.")
75
76
77 start()
```

Co powinieneś zobaczyć

A teraz zagram w tę grę.

Ćwiczenie 35. — sesja

```
$ python3.6 ex35.py
Znajdujesz się w mrocznym pokoju.
Po lewej i po prawej znajdują się drzwi.
Które wybierasz?
> lewe
Jest tutaj niedźwiedź.
Niedźwiedź ma beczkę miodu.
Ten gruby niedźwiedź siedzi przed następnymi drzwiami.
Jak przesuniesz niedźwiedzia?
> śmieję się z niedźwiedzia
```

```
Niedźwiedź odsunął się od drzwi.  
Możesz teraz przez nie przejść.  
> otwieram drzwi  
Ten pokój jest pełen złota. Ile złota zabierasz?  
> 1000  
Ty chciwy draniu! Dobra robota!
```

Zrób to sam

1. Narysuj mapę gry i zaznacz na niej sposób przejścia.
2. Napraw wszystkie błędy, w tym błędy ortograficzne.
3. Napisz komentarze do funkcji, których nie rozumiesz.
4. Rozwiń tę grę. Co możesz zrobić, aby ją jednocześnie uprościć i rozszerzyć?
5. Funkcja `gold_room` ma dziwny sposób na wpisanie liczby. Jakie są tu błędy? Czy możesz napisać to lepiej, niż ja to zrobiłem? Wskazówek poszukaj w sposobie działania `int()`.

Typowe pytania

Pomocy! Jak działa ten program?! Kiedy utkniesz w jakimś miejscu, próbując zrozumieć fragment kodu, po prostu nad każdą linią napisz komentarz wyjaśniający, co ona robi. Postaraj się, aby Twoje komentarze były krótkie i podobne do kodu. Następnie narysuj diagram pokazujący działanie kodu lub spróbuj to opisać w jednym akapicie. Jeśli to zrobisz, zrozumiesz kod.

Dlaczego napisałeś `while True`? To tworzy nieskończoną pętlę.

Do czego służy `exit(0)`? W wielu systemach operacyjnych działanie programu można przerwać poleceniem `exit(0)`, a przekazana liczba będzie wskazywać błąd lub brak błędu. Jeśli użyjesz `exit(1)`, będzie to oznaczać błąd, ale `exit(0)` jest „dobrym” wyjściem z programu. Jest to odwrotne do normalnej logiki boolowskiej (gdzie `0 == False`), żebyś mógł używać różnych liczb do wskazywania różnych błędnych wyników. Możesz na przykład użyć `exit(100)` dla innego błędu niż `exit(2)` lub `exit(1)`.

Dlaczego `input()` jest czasami zapisywane jako `input('> ')`? Parametrem funkcji `input` jest łańcuch znaków, który ma być wyświetlany jako znak zachęty przed uzyskaniem danych wejściowych od użytkownika.

Projektowanie i debugowanie

Skoro poznałeś już instrukcję `if`, zapoznam Cię z kilkoma regułami dotyczącymi stosowania pętli `for` i `while`, dzięki którym nie wpadniesz w kłopoty. Dam Ci też kilka wskazówek na temat debugowania, żebyś mógł rozwiązywać problemy z Twoim programami. Na koniec zaprojektujesz małą grę, podobną do tej z poprzedniego ćwiczenia, ale z drobną zmianą.

Zasady dotyczące instrukcji `if`

1. Każda instrukcja `if` musi zawierać `else`.
2. Jeśli to `else` nigdy nie będzie uruchamiane, ponieważ nie ma sensu, musisz użyć w nim funkcji `die`, która wypisuje komunikat o błędzie i przerywa program, tak jak robiliśmy w ostatnim ćwiczeniu. W ten sposób znajdziesz *wiele* błędów.
3. Nigdy nie zagnieżdżaj instrukcji `if` głębiej niż na dwa poziomy i zawsze staraj się, żeby był to tylko jeden poziom zagnieżdżenia.
4. Traktuj instrukcje `if` jak akapity, gdzie każda grupa `if-elif-else` jest podobna do zestawu zdań. Umieszczaj puste linie przed i po.
5. Twoje testy logiczne powinny być proste. Jeśli są złożone, przenieś ich obliczenia do zmiennych umieszczonych we wcześniej części funkcji i używaj dobrych nazw dla tych zmiennych.

Jeżeli zastosujesz się do tych prostych zasad, zaczniesz pisać kod lepszy od kodu większości programistów. Wróć do poprzedniego ćwiczenia i sprawdź, czy przestrzegałem tych wszystkich zasad. Jeśli nie, napraw moje błędy.

OSTRZEŻENIE! Nigdy nie bądź niewolnikiem zasad w prawdziwym życiu. Do celów szkoleniowych powinieneś przestrzegać tych reguł, żeby wzmocnić umysł, ale w prawdziwym życiu są one czasem po prostu głupie. Jeśli uważasz, że jakaś reguła jest głupia, spróbuj się do niej nie stosować.

Zasady dotyczące pętli

1. Używaj pętli `while` tylko do zapętlenia w nieskończoność, a to oznacza, że prawdopodobnie nie będziesz z niej korzystać. Dotyczy to tylko Pytona. W pozostałych językach wygląda to inaczej.
2. Używaj pętli `for` dla wszystkich pozostałych rodzajów zapętlenia szczególnie wtedy, kiedy istnieje stała lub ograniczona liczba rzeczy do zapętlenia.

Wskazówki dotyczące debugowania

1. Nie używaj „debuggera”. Debugger to jak tomografia całego ciała chorej osoby. Nie uzyskasz żadnych konkretnych użytecznych informacji, a otrzymasz całe mnóstwo danych, które nie pomagają i są po prostu mylące.
2. Najlepszym sposobem debugowania programu jest użycie polecenia `print`, aby wydrukować wartości zmiennych w określonych punktach programu i zobaczyć, gdzie dzieje się coś złego.
3. Gdy pracujesz nad fragmentami programu, od razu upewnij się, że działają prawidłowo. Nie pisz olbrzymich plików kodu, zanim spróbujesz je uruchomić. Koduj po trochu, uruchamiaj po trochu i naprawiaj po trochu.

Praca domowa

Teraz napisz grę podobną do tej, którą utworzyłem w poprzednim ćwiczeniu. Może to być jakakolwiek gra w podobnym gatunku. Poświęć na to cały tydzień i postaraj się, żeby była jak najbardziej interesująca. Do ćwiczeń „Zrób to sam” używaj w możliwie szerokim zakresie list, funkcji i modułów (pamiętasz te z ćwiczenia 13.?) i postaraj się znaleźć jak najwięcej nowych elementów Pythona, aby gra zadziałała.

Zanim zaczniesz kodować, musisz narysować mapę swojej gry. Najpierw na papierze buduj pokoje, potwory i pułapki, które gracz musi przejść.

Gdy już będziesz miał mapę, spróbuj ją zakodować. Jeśli napotkasz problemy z mapą, dostosuj ją i dopasuj kod.

Najlepiej pracować nad częścią oprogramowania w małych fragmentach według poniższych wskazówek.

1. Na kartce papieru lub fiszce spisz listę zadań, które musisz wykonać, aby ukończyć oprogramowanie. To jest Twoja lista rzeczy do zrobienia.
2. Wybierz ze swojej listy najłatwiejszą rzecz.
3. W pliku źródłowym napisz komentarze jako wskazówki dotyczące sposobu wykonania danego zadania w Twoim kodzie.
4. Pod komentarzami napisz część kodu.
5. Szybko uruchom skrypt, aby sprawdzić, czy ten kod zadziałał.
6. Kontynuuj pracę w cyklach: pisanie kodu, uruchamianie w celu przetestowania i naprawianie go, dopóki wszystko nie zadziała prawidłowo.
7. Wykreśl to zadanie z listy, a następnie wybierz następne najprostsze zadanie i powtórz kroki.

Ten proces pomoże Ci pracować nad oprogramowaniem w metodyczny i spójny sposób. Podczas pracy aktualizuj swoją listę, usuwając zadania, których naprawdę nie potrzebujesz, i dodając te, które są wymagane.

Przegląd symboli

Nadszedł czas na przegląd symboli i słów Pythona, które już znasz, oraz wybranie kilku nowych do kolejnych lekcji. Wypisałem wszystkie symbole i słowa kluczowe Pythona, które są ważne.

W tej lekcji postaraj się najpierw napisać z pamięci, co robi każde słowo. Następnie wyszukaj je w internecie i sprawdź, czy miałeś rację. Może to być trudne, ponieważ niektóre z nich niełatwo wyszukać, ale mimo to spróbuj.

Jeśli źle opiszesz z pamięci jakieś słowo lub symbol, sporządź dla niego fiszkę z poprawną definicją i spróbuj „naprawić” swoją pamięć.

Na koniec użyj każdego z nich (lub tylu, ilu dasz radę) w niewielkim programie Pythona. W ten sposób dowiesz się, co robi dany symbol, upewnij się, że dobrze go rozumiesz, poprawisz się, jeśli miałeś błędne pojęcie, a następnie użyjesz go, aby utrwalić w pamięci.

Słowa kluczowe

Słowo kluczowe	Opis	Przykład
and	Logiczna koniunkcja.	True and False == False
as	Część instrukcji with-as.	with X as Y: pass
assert	Przyjmuje (zapewnia), że coś jest prawdą.	assert False, "Error!"
break	Natychmiast zatrzymuje pętlę.	while True: break
class	Definiuje klasę.	class Person(object)
continue	Nie przetwarzaj dalszej części pętli, zrób to jeszcze raz.	while True: continue
def	Definiuje funkcję.	def X(): pass
del	Usuwa ze słownika.	del X[Y]
elif	Warunek else-if.	if: X; elif: Y; else: J
else	Warunek else.	if: X; elif: Y; else: J
except	Jeśli wystąpi wyjątek, zrób to.	except ValueError, e: print(e)
exec	Uruchom łańcuch znaków tak jak Python.	exec 'print("witaj")'
finally	Wyjątek czy nie, na koniec zrób to bez względu na wszystko.	finally: pass
for	Wykonuje pętlę przez kolekcję elementów.	for X in Y: pass
from	Importuje określone części modułu.	from x import Y
global	Deklaruje, że zmienna ma być globalna.	global X

Słowo kluczowe	Opis	Przykład
if	Warunek if.	if: X; elif: Y; else: J
import	Importuje moduł, który ma być użyty.	import os
in	Część pętli for. Również test X in Y.	for X in Y: pass also 1 in [1] == True
is	Tak jak ==, służy do testowania równości.	1 is 1 == True
lambda	Tworzy krótką funkcję anonimową.	s = lambda y: y ** y; s(3)
not	Logiczna negacja.	not True == False
or	Logiczna alternatywa.	True or False == True
pass	Ten blok jest pusty.	def empty(): pass
print	Drukuje łańcuch znaków	print('łańcuch_znaków')
raise	Rzuca wyjątek, gdy coś pójdzie źle.	raise ValueError("Nie")
return	Kończy działanie funkcji, zwracając wartość.	def X(): return Y
try	Spróbuj wykonać ten blok, a jeśli wystąpi wyjątek, przejdź do except.	try: pass
while	Pętla while.	while X: pass
with	Instrukcja menedżera kontekstu będąca alternatywą dla zwykłego zastosowania try/finally.	with X as Y: pass
yield	Zatrzymaj tu i zwróć sterowanie do podmiotu wywołującego.	def X(): yield Y; X().next()

Typy danych

W przypadku typów danych zapisz to, co tworzy każdy z nich. Jeśli chodzi na przykład o łańcuch znaków, napisz, jak tworzy się łańcuch znaków. W przypadku liczb zapisz kilka liczb.

Typ danych	Opis	Przykład
True	Prawda logiczna.	True or False == True
False	Fałsz logiczny.	False and True == False
None	Reprezentuje „nic” lub „brak wartości”.	x = None
bytes	Przechowuje bajty, czyli na przykład tekst, PNG, plik i tak dalej.	x = b"witaj"
strings	Przechowuje informacje tekstowe.	x = "witaj"
numbers	Przechowuje liczby całkowite.	i = 100
floats	Przechowuje liczby dziesiętne.	i = 10.389
lists	Przechowuje listy elementów.	j = [1,2,3,4]
dicts	Przechowuje pary klucz-wartość.	e = {'x': 1, 'y': 2}

Sekwencje ucieczki

Użyj każdej sekwencji ucieczki w łańcuchu znaków, aby upewnić się, że wiesz, co robią.

Sekwencja ucieczki	Opis
\\	Lewy ukośnik
\'	Pojedynczy cudzysłów
\"	Podwójny cudzysłów
\a	Dzwonek
\b	Backspace
\f	Wysunięcie strony
\n	Nowa linia
\r	Powrót karetki
\t	Tabulator
\v	Tabulator pionowy

Formatowanie łańcuchów znaków w starym stylu

To samo dotyczy kodów formatowania łańcuchów znaków: użyj ich w łańcuchach znaków, aby dowiedzieć się, co robią. Starszy kod Pythona 2 wykorzystuje te znaki formatujące do tego samego, do czego służą sformatowane łańcuchy znaków (ang. *f-strings*). Wypróbuj je jako alternatywę.

Kod formatowania	Opis	Przykład
%d	Dziesiętne liczby całkowite (bez miejsc po przecinku)	"%d" % 45 == '45'
%i	To samo, co %d	"%i" % 45 == '45'
%o	Liczba ósemkowa	"%o" % 1000 == '1750'
%u	Liczba dziesiętna bez znaku	"%u" % -1000 == '-1000'
%x	Liczba szesnastkowa (małą literą)	"%x" % 1000 == '3e8'
%X	Liczba szesnastkowa (wielką literą)	"%X" % 1000 == '3E8'
%e	Postać wykładnicza, małe „e”	"%e" % 1000 == '1.000000e+03'
%E	Postać wykładnicza, wielkie „E”	"%E" % 1000 == '1.000000E+03'
%f	Liczba rzeczywista zmiennoprzecinkowa	"%f" % 10.34 == '10.340000'
%F	To samo, co %f	"%F" % 10.34 == '10.340000'
%g	%f albo %e, w zależności, co jest krótsze	"%g" % 10.34 == '10.34'
%G	To samo, co %g, ale wielkimi literami	"%G" % 10.34 == '10.34'

Kod formatowania	Opis	Przykład
%c	Konwersja na znak	"%c" % 34 == ' "'
%r	Format łańcucha znaków (wykorzystujący funkcję repr()), format debugowania)	"%r" % int == "<type 'int'>"
%s	Format łańcucha znaków	"%s tam" % 'hej' == 'hej tam'
%%	Znak procentów	"%g%" % 10.34 == '10.34%'

Operatory

Niektórych przedstawionych w tabeli operatorów możesz nie znać, ale i tak poszukaj informacji na ich temat. Dowiedz się, co robią, a jeśli nadal nie będziesz mógł ich rozgryźć, zostaw to na później.

Operator		
+		
-		
*	Mnożenie	2 * 4 == 8
**	Opis	Przykład
/	Dodawanie	2 + 4 == 6
//	Odejmowanie	2 - 4 == -2
%	Interpolacja łańcuchów znaków lub operacja modulo	2 % 4 == 2
<	Mniejsze niż	4 < 4 == False
>	Większe niż	4 > 4 == False
<=	Mniejsze lub równe	4 <= 4 == True
>=	Większe lub równe	4 >= 4 == True
==	Równa się	4 == 5 == False
!=	Nie równa się	4 != 5 == True
()	Nawiasy	len('hej') == 3
[]	Nawiasy kwadratowe list	[1,3,4]
{ }	Nawiasy klamrowe słowników	{'x': 5, 'y': 10}
@	Małpa (dekoratory)	@classmethod
,	Przecinek	range(0, 10)
:	Dwukropek	def X():
.	Kropka	self.x = 10
=	Równość przypisania	x = 10
;	Średnik	print("hej"); print("tam")
+=	Dodaj i przypisz	x = 1; x += 2

Operator		
<code>--</code>	Odejmij i przypisz	<code>x = 1; x -= 2</code>
<code>*</code>	Pomnóż i przypisz	<code>x = 1; x *= 2</code>
<code>/</code>	Podziel i przypisz	<code>x = 1; x /= 2</code>
<code>//</code>	Podziel bez reszty i przypisz	<code>x = 1; x //= 2</code>
<code>%</code>	Wykonaj operację modulo i przypisz	<code>x = 1; x %= 2</code>
<code>**</code>	Wykonaj potęgowanie i przypisz	<code>x = 1; x **= 2</code>

Poświęć na to mniej więcej tydzień, ale jeśli skończysz szybciej, świetnie. Chodzi o to, żebyś spróbował zrozumieć wszystkie te symbole i zakodował je sobie w pamięci. Ważne jest również, abyś dowiedział się, czego *nie wiesz*, dzięki czemu będziesz mógł uzupełnić tę wiedzę później.

Czytanie kodu

Teraz znajdź jakiś kod Pythona do przeczytania. Powinieneś czytać każdy dostępny kod Pythona i próbować podkraść pomysły, które tam znajdziesz. Właściwie powinieneś mieć wystarczającą wiedzę, aby czytać kod, ale niekoniecznie będziesz rozumiał, co robi. Ta lekcja uczy Cię, jak rzeczy, które już opanowałeś, możesz wykorzystać do zrozumienia kodu napisanego przez kogoś innego.

Po pierwsze, wydrukuj kod, który chcesz zrozumieć. Tak, wydrukuj, ponieważ Twoje oczy i mózg są bardziej przyzwyczajone do czytania z kartki niż z ekranu komputera. Wydrukuj kilka stron na raz.

Po drugie, przejrzyj wydruk i zanotuj następujące rzeczy.

1. Funkcje i to, co robią.
2. Gdzie każdej zmiennej po raz pierwszy nadawana jest wartość.
3. Wszystkie zmienne o tych samych nazwach w różnych częściach programu. Mogą one później okazać się problematyczne.
4. Wszystkie instrukcje `if` bez klauzul `else`. Czy są poprawne?
5. Wszystkie pętle `while`, które mogą się nie kończyć.
6. Wszystkie części kodu, których nie możesz zrozumieć z dowolnego powodu.

Po trzecie, gdy już wszystko zaznaczysz, spróbuj objaśnić to sobie, pisząc na bieżąco komentarze. Opisz funkcje, sposób ich użycia, wykorzystane zmienne i wszystko, co możesz zrozumieć z tego kodu.

Na końcu we wszystkich trudnych częściach kodu prześledź wartości każdej zmiennej linia po linii, funkcja po funkcji. Wydrukuj to jeszcze raz i napisz na marginesie wartość każdej zmiennej, którą musisz „prześledzić”.

Gdy będziesz miał już dość dobre wyobrażenie o tym, co robi kod, wróć do komputera i przeczytaj go ponownie, aby sprawdzić, czy znajdziesz jakieś nowe rzeczy. Szukaj cały czas nowych kodów i postępuj z nimi w ten sam sposób dopóki, dopóty nie będziesz już potrzebował wydruków.

Zrób to sam

1. Dowiedz się, co to jest „schemat blokowy” i narysuj kilka takich schematów.
2. Jeśli znajdziesz błędy w czytany kodzie, spróbuj je naprawić i wyślij autorowi swoje zmiany.
3. Inną techniką, w której nie używasz papierowych wydruków, jest umieszczanie komentarzy z notatkami w kodzie. Czasami mogą one stać się rzeczywistymi komentarzami, które pomogą następnej osobie.

Typowe pytania

Jak wyszukiwać te rzeczy w internecie? Po prostu umieść ciąg `python3.6` przed dowolną wyszukiwaną frazą, aby na przykład znaleźć `yield`, wpisz **`python3.6 yield`**.

Operacje na listach

Poznałeś już listy. Gdy uczyłeś się o pętlach `while`, na końcu listy „dołączałeś” liczby i drukowałeś je. Były też ćwiczenia „Zrób to sam”, w których Twoim zadaniem było wyszukiwanie w dokumentacji Pythona wszystkich pozostałych rzeczy, jakie można robić z listami. Było to już chwilę temu, więc jeśli nie wiesz, o czym mówię, przejrzyj jeszcze raz te tematy.

Znalazłeś? Pamiętasz? Dobrze. Gdy wykonywałeś to ćwiczenie, miałeś listę, na której „wywoływałeś” funkcję `append`. Być może jednak nie do końca rozumiesz, o co w tym wszystkim chodzi, więc zobaczmy, co możemy zrobić z listami.

Kiedy piszesz `mystuff.append('witaj')`, w rzeczywistości inicjujesz wewnątrz Pythona łańcuch zdarzeń, który ma spowodować, że coś się stanie z listą `mystuff`. A tak to działa.

1. Python zauważa, że wspomniałeś o `mystuff` i szuka tej zmiennej. Być może będzie musiał wykonać przeszukiwanie wstecz, aby sprawdzić, czy utworzyłeś ją za pomocą `=`, czy jest to argument funkcji lub jest to zmienna globalna. Tak czy inaczej, najpierw musi znaleźć `mystuff`.
2. Po znalezieniu `mystuff` Python odczytuje operator `.` (kropkę) i zaczyna przeglądać *zmiennne*, które są częścią `mystuff`. Ponieważ `mystuff` jest listą, Python „wie”, że posiada ona wiele funkcji.
3. Następnie Python natrafia na `append` i porównuje tę nazwę ze wszystkimi nazwami, które `mystuff` uznaje za własne. Jeśli `append` do nich należy (a należy), Python pobiera ją, aby jej użyć.
4. Następnie Python rejestruje znak `(` (nawias otwierający) i uświadamia sobie: „Hej, to powinna być funkcja”. W tym momencie *wywołuje* (uruchamia lub wykonuje) funkcję, tak jak zwykle, ale wywołuje ją z *dodatkowym* argumentem.
5. Ten *dodatkowy* argument to... `mystuff`! Wiem, to dziwne, prawda? Jednak tak działa Python, więc najlepiej po prostu to zapamiętać i przyjąć, że taki jest rezultat. W efekcie na koniec tego wszystkiego otrzymujemy wywołanie funkcji, które w rzeczywistości wygląda tak: `append(mystuff, 'witaj')`, a nie tak jak można przeczytać w kodzie: `mystuff.append('witaj')`.

W większości przypadków nie musisz wiedzieć, że dzieją się właśnie takie rzeczy, ale jest to pomocne, gdy otrzymasz od Pythona komunikat o błędzie, taki jak ten:

```
$ python3.6
>>> class Thing(object):
...     def test(message):
...         print(message)
...
>>> a = Thing()
>>> a.test("witaj")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: test() takes exactly 1 argument (2 given)
>>>
```

Co to było? Cóż, wpisałem w powłocę Pythona pewien kod i pokazałem Ci trochę magii. Nie widziałeś jeszcze `class`, ale zajmiemy się tym później. Na razie widzisz, że Python poinformował, iż `test()` takes exactly 1 argument (2 given) (`test()` przyjmuje dokładnie 1 argument (podano 2)). Oznacza to, że Python zamienił `a.test("witaj")` na `test(a, "witaj")` oraz że ktoś popełnił gdzieś błąd i nie dodał argumentu dla `a`.

To dość dużo informacji na jeden raz, ale poświęcimy kilka ćwiczeń, żeby ta koncepcja utrwaliła Ci się w głowie. Na początek ćwiczenie, które miesza łańcuchy znaków i listy dla lepszej zabawy.

ex38.py

```
1  ten_things = "Jabłko Pomarańcze Wrony Telefon Światło Cukier"
2
3  print("Chwileczkę, na tej liście nie ma 10 rzeczy. Poprawmy to.")
4
5  stuff = ten_things.split(' ')
6  more_stuff = ["Dzień", "Noc", "Piosenka", "Frisbee",
7               "Kukurydza", "Banan", "Dziewczyna", "Chłopak"]
8
9  while len(stuff) != 10:
10     next_one = more_stuff.pop()
11     print("Dodawanie: ", next_one)
12     stuff.append(next_one)
13     print(f"Teraz jest {len(stuff)} elementów.")
14
15  print("Teraz to mamy: ", stuff)
16
17  print("Zróbmy jeszcze parę rzeczy ze stuff.")
18
19  print(stuff[1])
20  print(stuff[-1]) # ła! nieźle
21  print(stuff.pop())
22  print(' '.join(stuff)) # serio? super!
23  print('#'.join(stuff[3:5])) # super hiper!
```

Co powinieneś zobaczyć

Ćwiczenie 38. — sesja

```
Chwileczkę, na tej liście nie ma 10 rzeczy. Poprawmy to.
Dodawanie: Chłopak
Teraz jest 7 elementów.
Dodawanie: Dziewczyna
Teraz jest 8 elementów.
Dodawanie: Banan
```

```
Teraz jest 9 elementów.  
Dodawanie: Kukurydza  
Teraz jest 10 elementów.  
Teraz to mamy: ['Jabłko', 'Pomarańcze', 'Wrony', 'Telefon', 'Światło',  
                'Cukier', 'Chłopak', 'Dziewczyna', 'Banan', 'Kukurydza']  
Zróbmy jeszcze parę rzeczy ze stuff.  
Pomarańcze  
Kukurydza  
Kukurydza  
Jabłko Pomarańcze Wrony Telefon Światło Cukier Chłopak Dziewczyna Banan  
Telefon#Światło
```

Co mogą robić listy

Załóżmy, że chcesz utworzyć grę komputerową opartą na *Go Fish*. Jeśli nie wiesz, czym jest *Go Fish*, poświęć chwilę i poczytaj w internecie. Aby zrobić tę grę, będziesz potrzebował jakiegoś sposobu na zaimplementowanie w programie Pythona koncepcji „talii kart”. Następnie musisz napisać kod, który popracuje z tą wyimaginowaną wersją talii kart, aby osoba grająca w Twoją grę miała wrażenie, że jest to prawdziwa talia. Potrzebna jest struktura „talii kart”, a programiści nazywają taką rzecz *strukturą danych*.

Czym jest struktura danych? Można powiedzieć, że **struktura danych** to po prostu formalny sposób na **strukturyzację** (uporządkowanie) pewnych **danych** (faktów). To naprawdę jest takie proste. Chociaż niektóre struktury danych mogą stać się niesamowicie skomplikowane, są jedynie metodą na przechowywanie faktów w programie, aby można było uzyskiwać do nich dostęp na różne sposoby. Struktury uporządkowują dane.

Listy są jedną z najczęstszych struktur danych używanych przez programistów. Lista to uporządkowana lista rzeczy, które chcesz przechowywać i udostępniać losowo lub liniowo według indeksu. Co?! Pamiętaj, co mówiłem: tylko dlatego, że programista powiedział: „Lista jest listą”, nie znaczy to, że jest bardziej skomplikowana niż to, czym jest lista w prawdziwym świecie. Spójrzmy na talię karty jak na przykład listy.

1. Masz kilkadziesiąt kart z wartościami.
2. Karty znajdują się na stosie, w liście lub liście od górnej karty do dolnej.
3. Możesz zdejmować karty z góry, z dołu lub wyjmować je losowo ze środka.
4. Jeśli chcesz znaleźć konkretną kartę, musisz wziąć talię i przejść przez nią karta po karcie.

Spójrzmy na to, co powiedziałem.

„**Uporządkowana lista...**” — tak, talia kart jest uporządkowana i posiada pierwszą oraz ostatnią kartę.

„**...rzeczy, które chcesz przechowywać...**” — tak, karty to rzeczy, które chcę przechowywać.

„**...i udostępniać losowo...**” — tak, mogę wyciągnąć kartę z dowolnego miejsca z talii.

„**...lub liniowo...**” — tak, jeśli chcę znaleźć konkretną kartę, mogę zacząć od początku i iść w kolejności.

„...według indeksu.” — prawie, ponieważ gdybym z talii kart kazał Ci wyciągnąć kartę o indeksie 19, musiałbyś liczyć po kolei, dopóki nie doszedłbyś do wskazanej karty. W naszych listach Pythona komputer może po prostu przeskoczyć bezpośrednio do dowolnego indeksu, który podamy.

To wszystko, co robi lista. Powinno pozwolić Ci to zrozumieć tę koncepcję w programowaniu. Każda koncepcja w programowaniu zwykle ma jakieś odniesienie do świata rzeczywistego. A przynajmniej te użyteczne koncepcje. Jeśli potrafisz znaleźć analogię w świecie rzeczywistym, możesz jej użyć, aby zrozumieć, jakie możliwości ma struktura danych.

Kiedy używać list

Z listy korzystasz za każdym razem, gdy masz coś, do czego możesz zastosować przydatne funkcjonalności struktury danych listy.

1. Jeśli chcesz utrzymać porządek. Pamiętaj, że jest to porządek *wylistowany*, a nie *posortowany*. Listy nie przeprowadzają sortowania.
2. Jeśli chcesz uzyskać dostęp do zawartości losowo według liczby. Pamiętaj, że używasz liczb *kardynalnych* zaczynających się od 0.
3. Jeśli musisz przejrzeć przez zawartość liniowo (od pierwszego do ostatniego elementu). Pamiętaj, że właśnie po to są pętle `for`.

Tak jest, gdy używasz listy.

Zrób to sam

1. Weź każdą wywoływaną funkcję i przejdź przez czynności wywołania funkcji, aby je przetłumaczyć na to, co robi Python. Przykładowo `more_stuff.pop()` to `pop(more_stuff)`.
2. Przetłumacz te dwa sposoby zapisywania wywołania funkcji na język polski. Przykładowo `more_stuff.pop()` czytamy: „Wywołaj `pop` na `more_stuff`”. Natomiast `pop(more_stuff)` oznacza: „Wywołaj `pop` z argumentem `more_stuff`”. Postaraj się zrozumieć, dlaczego naprawdę są to te same rzeczy.
3. Poczytaj w internecie na temat programowania obiektowego. Skonfundowany? Ja też tak się czułem. Nie przejmuj się. Nauczysz się wystarczająco dużo, aby być niebezpiecznym i później będziesz mógł powoli nauczyć się więcej.
4. Przeczytaj, czym jest klasa w Pythonie. *Nie czytaj o tym, jak inne języki używają słowa „klasa”*. To Ci tylko zamiesza w głowie.
5. Nie przejmuj się, jeśli nie masz pojęcia, o czym mówię. Programiści lubią się bawić w inteligentów, więc wymyślili programowanie obiektowe (ang. *object-oriented programming*), nazwali je OOP, a potem używali go w nadmiarze. Jeśli uważasz, że to trudne, powinieneś spróbować „programowania funkcyjnego”.
6. Znajdź 10 przykładów rzeczy w prawdziwym świecie, które pasowałyby do listy. Spróbuj napisać kilka skryptów, aby z nimi popracować.

Typowe pytania

Czy nie mówiłeś, żeby nie używać pętli `while`? Tak, ale pamiętaj, że czasami możesz złamać zasady, jeśli masz dobry powód. Tylko idioci są niewolnikami reguł przez cały czas.

Dlaczego `join(' ', stuff)` nie działa? Dokumentacja dla `join` została napisana w sposób bezsensowny. To nie działa w ten sposób, gdyż jest to metoda wywoływana na łańcuchu znaków **wstawianym** pomiędzy łączonymi elementami listy. Przepisz to tak: `' '.join(stuff)`.

Dlaczego użyłeś pętli `while`? Spróbuj przepisać skrypt za pomocą pętli `for` i sprawdź, czy jest to łatwiejsze.

Co robi `stuff[3:5]`? Wyodrębnia z listy `stuff` „wycinek” od 3. do 4. elementu, co oznacza, że *nie* zawiera elementu 5. Jest podobne do działania `range(3,5)`.

Słowniki, piękne słowniki

Poznasz teraz strukturę danych Pythona zwaną słownikiem (ang. *dictionary*). Słownik jest sposobem przechowywania danych, podobnie jak lista, ale zamiast stosowania jedynie liczb do uzyskiwania danych, można używać prawie wszystkiego. Dzięki temu możesz traktować słownik jak bazę danych do przechowywania i porządkowania danych.

Porównajmy możliwości słowników i list. Na to pozwala lista.

Ćwiczenie 39. — sesja Pythona

```
>>> things = ['a', 'b', 'c', 'd']
>>> print(things[1])
b
>>> things[1] = 'z'
>>> print(things[1])
z
>>> things
['a', 'z', 'c', 'd']
<<listing-koniec>>
```

Do **indeksowania** listy możesz używać liczb, co oznacza, że za pomocą liczb możesz dowiedzieć się, co znajduje się na liście. To już powinieneś wiedzieć, ale upewnij się, że rozumiesz, iż do pobierania elementów z listy możesz używać *tylko* liczb.

Słownik pozwala korzystać z *czegośkolwiek*, nie tylko z liczb. Tak, słownik kojarzy jedną rzecz z drugą, bez względu na to, czym są. Spójrz niżej.

Ćwiczenie 39. — sesja Pythona

```
>>> stuff = {'name': 'Zed', 'age': 39, 'height': 6 * 12 + 2}
>>> print(stuff['name'])
Zed
>>> print(stuff['age'])
39
>>> print(stuff['height'])
74
>>> stuff['city'] = "SF"
>>> print(stuff['city'])
SF
```

Zobaczysz, że zamiast zwykłych liczb używamy łańcuchów znaków, aby powiedzieć, czego chcemy ze słownika `stuff`. Za pomocą łańcuchów znaków możemy również wstawiać nowe rzeczy do słownika. Jednak nie muszą być to koniecznie łańcuchy znaków. Możemy również zrobić tak.

Ćwiczenie 39. — sesja Pythona

```
>>> stuff[1] = "Łań"
>>> stuff[2] = "Czadowo"
>>> print(stuff[1])
Łań
>>> print(stuff[2])
Czadowo
```

W tym kodzie użyłem liczb, a potem — kiedy to drukuję — widzieć, że mamy w słowniku liczby jako klucze i łańcuchy znaków jako wartości. Mogłbym użyć czegokolwiek. No prawie, ale na razie udawajmy, że możemy użyć czegokolwiek.

Oczywiście słownik, do którego można tylko wstawiać różne rzeczy, jest niezbyt użyteczny, więc poniżej pokazałem, jak usuwać te rzeczy za pomocą słowa kluczowego `del`.

Ćwiczenie 39. — sesja Pythona

```
>>> del stuff['city']
>>> del stuff[1]
>>> del stuff[2]
>>> stuff
{'name': 'Zed', 'age': 39, 'height': 74}
```

Przykład słownika

Wykonamy teraz ćwiczenie, które *musisz przestudiować* bardzo uważnie. Chcę, żebyś wpisał ten kod i spróbował zrozumieć, co się w nim dzieje. Zwróć uwagę na umieszczanie rzeczy w słowniku i pobieranie ich za pomocą skrótów oraz na wszystkie użyte operacje. Przyjrzyj się, jak ten przykład mapuje stany na ich skróty, a następnie skróty na miasta w stanach. Pamiętaj, że **mapowanie** (inaczej **odwzorowanie**) jest kluczową koncepcją w słowniku.

ex39.py

```
1  # tworzymy mapowanie nazwy stanu na skrót
2  states = {
3      'Oregon': 'OR',
4      'Floryda': 'FL',
5      'Kalifornia': 'KA',
6      'Nowy Jork': 'NJ',
7      'Michigan': 'MI'
8  }
9
10 # tworzymy podstawowy zestaw stanów i znajdujących się w nich miast
11 cities = {
12     'KA': 'San Francisco',
13     'MI': 'Detroit',
14     'FL': 'Jacksonville'
15 }
16
17 # dodajemy kilka miast
```

```
18 cities['NJ'] = 'Nowy Jork'
19 cities['OR'] = 'Portland'
20
21 # drukujemy kilka miast
22 print('-' * 10)
23 print("Stan NJ ma: ", cities['NJ'])
24 print("Stan OR ma: ", cities['OR'])
25
26 # drukujemy kilka stanów
27 print('-' * 10)
28 print("Skrót dla Michigan to: ", states['Michigan'])
29 print("Skrót dla Floryda to: ", states['Floryda'])
30
31 # używamy najpierw słownika state, a potem cities
32 print('-' * 10)
33 print("Michigan ma: ", cities[states['Michigan']])
34 print("Floryda ma: ", cities[states['Floryda']])
35
36 # drukujemy skrót każdego stanu
37 print('-' * 10)
38 for state, abbrev in list(states.items()):
39     print(f"{state} ma skrót {abbrev}")
40
41 # drukujemy każde miasto w stanie
42 print('-' * 10)
43 for abbrev, city in list(cities.items()):
44     print(f"{abbrev} ma miasto {city}")
45
46 # teraz obie te rzeczy naraz
47 print('-' * 10)
48 for state, abbrev in list(states.items()):
49     print(f"Stan {state} ma skrót {abbrev}")
50     print(f"oraz miasto {cities[abbrev]}")
51
52 print('-' * 10)
53 # bezpiecznie pobieramy skrót według nazwy stanu, który może nie istnieć
54 state = states.get('Texas')
55
56 if not state:
57     print("Przepraszamy, nie ma stanu Texas.")
58
59 # pobieramy miasto za pomocą domyślnej wartości
60 city = cities.get('TX', 'Nie istnieje')
61 print(f"Miasto dla stanu 'TX' to: {city}")
```

Co powinieneś zobaczyć

Ćwiczenie 39. — sesja

```
$ python3.6 ex39.py
```

```
-----
```

```
Stan NJ ma: Nowy Jork
```

```

Stan OR ma:  Portland
-----
Skrót dla Michigan to:  MI
Skrót dla Floryda to:  FL
-----
Michigan ma:  Detroit
Floryda ma:  Jacksonville
-----
Oregon ma skrót OR
Floryda ma skrót FL
Kalifornia ma skrót KA
Nowy Jork ma skrót NJ
Michigan ma skrót MI
-----
KA ma miasto San Francisco
MI ma miasto Detroit
FL ma miasto Jacksonville
NJ ma miasto Nowy Jork
OR ma miasto Portland
-----
Stan Oregon ma skrót OR
oraz miasto Portland
Stan Floryda ma skrót FL
oraz miasto Jacksonville
Stan Kalifornia ma skrót KA
oraz miasto San Francisco
Stan Nowy Jork ma skrót NJ
oraz miasto Nowy Jork
Stan Michigan ma skrót MI
oraz miasto Detroit
-----
Przepraszamy, nie ma stanu Texas.
Miasto dla stanu 'TX' to: Nie istnieje

```

Co mogą robić słowniki

Słownik jest kolejnym przykładem struktury danych i — podobnie jak lista — jest jedną ze struktur danych najczęściej używanych w programowaniu. Słownik służy do *mapowania* lub *kojarzenia* rzeczy, które chcesz przechowywać, z kluczami potrzebnymi do ich pobierania. Programiści nie używaliby terminu, takiego jak „słownik”, dla czegoś, co nie działa jak prawdziwy słownik pełen słów, więc użyjmy tegoż jako naszego przykładu ze świata realnego.

Powiedzmy, że chcesz dowiedzieć się, co oznacza słowo „*honorificabilitudinitatibus*”. Dzisiaj po prostu skopiowałbyś to słowo i wkleiłbyś do wyszukiwarki, a następnie otrzymał odpowiedź. Możemy powiedzieć, że wyszukiwarka jest jak naprawdę wielka, ogromnie złożona wersja *Słownika Oksfordzkiego* (ang. *Oxford English Dictionary* — **OED**). W czasach przed powstaniem wyszukiwarek wykonałbyś następujące czynności.

1. Poszedłbyś do swojej biblioteki i wypożyczył „słownik”. Powiedzmy, że to OED.
2. Wiesz, że słowo „*honorificabilitudinitatibus*” zaczyna się na literę „H”, więc poszukałbyś na brzegu książki zakładki oznaczonej literą „H”.

3. Przekartkowałbyś strony, aż znalazłbyś się blisko miejsca, gdzie zaczynają się słowa na „hon”.
4. Przekartkowałbyś jeszcze kilka stron, aż znalazłbyś „honorificabilitudinitatibus” lub dotarł do początku słów na „hp” i zdał sobie sprawę, że tego słowa nie ma w OED.
5. Po znalezieniu wpisu przeczytałbyś definicję, aby dowiedzieć się, co znaczy to słowo.

Prawie dokładnie tak działa słownik w Pythonie i zasadniczo „mapujesz” słowo „honorificabilitudinitatibus” na jego definicję. Słownik w Pythonie jest po prostu jak słownik w świecie rzeczywistym, na przykład *Słownik Oksfordzki*.

Zrób to sam

1. Wykonaj ten sam rodzaj mapowania z miastami i województwami (regionami) w swoim kraju lub w jakimś innym.
2. Znajdź dokumentację Pythona dla słowników i spróbuj zrobić z nimi jeszcze więcej rzeczy.
3. Dowiedz się, czego *nie możesz zrobić* ze słownikami. Jedną z ważniejszych rzeczy jest to, że słowniki nie są uporządkowane, więc spróbuj się tym pobawić.

Typowe pytania

Jaka jest różnica między listą a słownikiem? Lista jest uporządkowana. Słownik służy do dopasowywania pewnych elementów (nazywanych „kluczami”) do innych elementów (nazywanych „wartościami”).

Kiedy mógłbym użyć słownika? W sytuacji, kiedy musisz wziąć jedną wartość i „wyszukać” inną wartość. Właściwie można by nazywać słowniki „tablicami wyszukiwania”.

Do czego mógłbym użyć listy? Używaj listy dla każdej sekwencji rzeczy, które muszą być uporządkowane i których wyszukujesz tylko za pomocą indeksu liczbowego.

Co zrobić, jeśli potrzebuję słownika, ale chcę, żeby był uporządkowany? Przyjrzyj się strukturze danych `collections.OrderedDict` w Pythonie. Poszukaj dokumentacji w internecie.

Moduły, klasy i obiekty

Python jest nazywany „językiem programowania obiektowego”. Oznacza to, że w Pythonie istnieje konstrukcja zwana **klasą**, która pozwala w szczególny sposób strukturyzować oprogramowanie. Za pomocą klas można dodać spójność do programów, aby można było ich używać w bardziej przejrzysty sposób. Przynajmniej taka jest teoria.

Wykorzystując to, co już wiesz o słownikach i modułach, przedstawię podstawy dotyczące programowania obiektowego, klas i obiektów. Mój problem polega na tym, że dla mnie programowanie obiektowe (ang. *object-oriented programming* — **OOP**) jest po prostu dziwne. Muszę się z tym zmierzyć, spróbować zrozumieć, co tutaj piszę, wpisać kod, a w następnym ćwiczeniu wbiję to sobie do głowy.

No to zaczynamy.

Moduły są jak słowniki

Wiesz, w jaki sposób tworzony i używany jest słownik oraz że jest to sposób na odwzorowanie jednej rzeczy na drugą. Oznacza to, że jeśli masz słownik z kluczem `apple` (jabłko) i chcesz otrzymać jego wartość, musisz wykonać poniższe czynności.

ex40a.py

```
1  mystuff = {'apple': "JA JESTEM JABŁKO!"}
2  print(mystuff['apple'])
```

Zapamiętaj tę koncepcję „pobierz X z Y”, a teraz pomyśl o modułach. Do tej pory utworzyłeś już kilka modułów i powinieneś znać następujące cechy charakterystyczne modułu.

1. Jest to plik Pythona z pewnymi funkcjami lub zmiennymi...
2. ...który można zaimportować...
3. ...i uzyskiwać dostęp do jego funkcji lub zmiennych za pomocą operatora `.` (kropki).

Wyobraź sobie, że mam moduł, który nazywam *mystuff.py*, i umieszczam w nim funkcję o nazwie `apple`. Oto moduł *mystuff.py*.

ex40a.py

```
1  # to znajduje się w mystuff.py
2  def apple():
3      print("JA JESTEM JABŁKO!")
```

Gdy mam ten kod, mogę użyć modułu *mystuff* za pomocą `import`, a następnie uzyskać dostęp do funkcji `apple`.

ex40a.py

```
1 import mystuff
2 mystuff.apple()
```

Mogę również umieścić w nim zmienną o nazwie tangerine (mandarynka).

ex40a.py

```
1 def apple():
2     print("JA JESTEM JABŁKO!")
3
4     # to jest po prostu zmienna
5     tangerine = "Żywe odbicie snu"
```

W ten sam sposób mogę uzyskać do niej dostęp.

ex40a.py

```
1 import mystuff
2
3 mystuff.apple()
4 print(mystuff.tangerine)
```

Jeśli przypomnisz sobie słowniki, powinieneś zauważyć, jak bardzo jest to podobne do używania słownika, ale składnia jest inna. Oto porównanie.

ex40a.py

```
1 mystuff['apple'] # pobranie apple ze słownika
2 mystuff.apple() # pobranie apple z modułu
3 mystuff.tangerine # to samo, to jest po prostu zmienna
```

Oznacza to, że mamy w Pythonie *bardzo powszechny wzorzec*.

1. Weź kontener typu klucz=wartość.
2. Wyciągnij coś z niego za pomocą nazwy klucza.

W przypadku słownika kluczem jest łańcuch znaków, a składnia to `[klucz]`. W przypadku modułu klucz jest identyfikatorem, a składnia to `.klucz`. Poza tym obie te rzeczy są prawie takie same.

Klasy są jak moduły

Możesz potraktować moduł jak wyspecjalizowany słownik, który może przechowywać kod Pythona, abyś mógł uzyskiwać do niego dostęp za pomocą operatora `.` (kropki). Python dysponuje jeszcze jedną konstrukcją, która służy do podobnego celu i nazywa się klasą. Klasa to sposób na grupowanie funkcji i danych oraz umieszczanie ich w kontenerze, abyś mógł uzyskiwać do nich dostęp za pomocą operatora `.` (kropki).

Gdybym miał utworzyć klasę podobną do modułu `mystuff`, zrobiłbym coś takiego.

ex40a.py

```
1 class MyStuff(object):
2
3     def __init__(self):
4         self.tangerine = "And now a thousand years between"
5
6     def apple(self):
7         print("JESTEM JABŁKO Z KLASĄ!")
```

Skomplikowane w porównaniu z modułami? Na pewno dużo więcej się tutaj dzieje, ale po-
winienes zrozumieć, dlaczego przypomina to „minimoduł” z klasą `MyStuff` zawierającą funkcję
`apple()`. Prawdopodobnie mylące są funkcja `__init__()` i użycie `self.tangerine` do ustawienia
instancji zmiennej `tangerine`.

Oto wyjaśnienie, dlaczego zamiast modułów używa się klas: możesz wziąć klasę `MyStuff` i użyć
jej do utworzenia wielu takich samych, milionów, jeśli chcesz, a żadna nie będzie kolidowała
z drugą. Gdy importujesz moduł, jest tylko jeden dla całego programu, chyba że zrobisz
jakąś potworną sztuczkę.

Jednak zanim to zrozumiesz, musisz dowiedzieć się, czym jest „obiekt” i jak pracować
z `MyStuff` podobnie, jak robisz to z modulem *mystuff.py*.

Obiekty są jak import

Jeśli klasa jest jak „minimoduł”, musi istnieć koncepcja podobna do importu (`import`), ale dla
klas. Koncepcja ta nazywa się „instancjonowaniem”, co jest po prostu wymyślnym, niezno-
śnym, przesadnie inteligentnym sposobem na powiedzenie „tworzenie”. Gdy instancjonujesz
klasę, otrzymujesz tak zwany obiekt.

Klasę instancjonujesz (tworzysz) poprzez wywołanie jej tak, jakby była funkcją, na przykład tak.

ex40a.py

```
1 thing = MyStuff()
2 thing.apple()
3 print(thing.tangerine)
```

Pierwsza linia to operacja „instancjonowania” (tworzenia instancji) i bardzo przypomina
wywoływanie funkcji. Jednak za kulisami Python koordynuje za nas sekwencję zdarzeń. Przejdę
przez te kroki, używając powyższego kodu dla `MyStuff`.

1. Python szuka `MyStuff()` i zauważa, że jest to klasa, którą zdefiniowałeś.
2. Python tworzy pusty obiekt ze wszystkimi funkcjami określonymi w tej klasie za
pomocą `def`.
3. Następnie Python sprawdza, czy utworzyłeś „magiczną” funkcję `__init__`, a jeśli
ją masz, wywołuje tę funkcję do zainicjowania nowo utworzonego pustego
obiektu.

4. Potem w funkcji `__init__` klasy `MyStuff` otrzymujesz tę dodatkową zmienną `self`, która jest pustym obiektem utworzonym dla Ciebie przez Pythona, i możesz ustawić w nim zmienne, tak jak zrobiłbyś to z modułem, słownikiem lub innym obiektem.
5. W tym przypadku ustawiasz `self.tangerine` na słowa piosenki i masz zainicjowany obiekt.
6. Teraz Python może wziąć ten nowo wygenerowany obiekt i przypisać go do zmiennej `thing`, z którą możesz pracować.

Oto podstawy tego, jak Python robi ten „miniimport”, gdy wywołujesz klasę jak funkcję. Zapamiętaj, że w ten sposób Python **nie daje** Ci klasy, ale w zamian używa jej jako **wzorca** do zbudowania kopii tego rodzaju rzeczy.

Pamiętaj, że podaję nieco niedokładne wyobrażenie, jak to działa, żebyś mógł zacząć rozumieć klasy na podstawie tego, co wiesz o modułach. Prawda jest taka, że w tym punkcie klasy i obiekty nagle zaczynają oddalać się od modułów. Gdybym był całkowicie szczerzy, powiedziałbym raczej coś takiego.

- Klasy są jak wzorce lub definicje do tworzenia nowych minimodułów.
- Instancjonowanie jest sposobem na utworzenie jednego z tych minimodułów **oraz** jednocześnie zaimportowanie go. „Instancjonowanie” oznacza po prostu tworzenie obiektu z klasy.
- Powstały minimodul nazywa się obiektem i przypisujesz go do zmiennej, aby można było z nim pracować.

Na tym etapie obiekty zachowują się inaczej niż moduły i powinno Ci to posłużyć jedynie jako pomost do zrozumienia klas i obiektów.

Różne sposoby pobierania elementów

Mam teraz trzy sposoby na pobieranie elementów z innych rzeczy.

ex40a.py

```
1  # w stylu słownikowym
2  mystuff['apples']
3
4  # w stylu modułowym
5  mystuff.apples()
6  print(mystuff.tangerine)
7
8  # w stylu klasowym
9  thing = MyStuff()
10 thing.apples()
11 print(thing.tangerine)
```

Pierwszy przykład klasy

Powinieneś zacząć dostrzegać podobieństwa w tych trzech typach kontenerów klucz=wartość i prawdopodobnie mieć mnóstwo pytań. Wstrzymaj się z pytaniami, ponieważ następne ćwiczenie łopatologicznie wyłoży całe „obiektowe słownictwo”. W tym ćwiczeniu chcę tylko, żebyś wpisał oraz uruchomił poniższy kod i nabrał trochę doświadczenia, zanim przejdiesz dalej.

ex40.py

```
1  class Song(object):
2
3      def __init__(self, lyrics):
4          self.lyrics = lyrics
5
6      def sing_me_a_song(self):
7          for line in self.lyrics:
8              print(line)
9
10 happy_bday = Song(["Happy birthday to you",
11                    "Nie chcę zostać pozwany",
12                    "Więc tutaj przerwę"])
13
14 bulls_on_parade = Song(["They rally around tha family",
15                         "With pockets full of shells"])
16
17 happy_bday.sing_me_a_song()
18
19 bulls_on_parade.sing_me_a_song()
```

Co powinieneś zobaczyć

Ćwiczenie 40. — sesja

```
$ python3.6 ex40.py
Happy birthday to you
Nie chcę zostać pozwany
Więc tutaj przerwę
They rally around tha family
With pockets full of shells
```

Zrób to sam

1. Napisz więcej piosenek, używając tej klasy, i upewnij się, że rozumiesz, iż jako słowa piosenki przekazujesz listę łańcuchów znaków.
2. Umieść słowa piosenki w osobnej zmiennej, a następnie przekaz tę zmienną do klasy, aby jej użyć.

3. Sprawdź, czy potrafisz zhakować ten kod i sprawić, żeby robił więcej rzeczy. Nie przejmuj się, jeśli nie masz pojęcia, jak to zrobić, po prostu spróbuj i zobacz, co się stanie. Popsuj go, zniszcz, przemieli, nie możesz go zranić.
4. Poszukaj w internecie informacji na temat programowania obiektowego i spróbuj przepełnić swój mózg tym, co przeczytasz. Nie przejmuj się, jeśli nie będzie miało to dla Ciebie żadnego sensu. Połowa tych rzeczy nie ma sensu również dla mnie.

Typowe pytania

Dlaczego potrzebuję `self`, gdy tworzę `__init__` lub inne funkcje dla klas? Jeśli nie masz `self`, wtedy kod, taki jak `cheese = 'Franek'`, jest niejednoznaczny. W tym kodzie niejasne jest to, czy masz na myśli atrybut `cheese` **instancji**, czy lokalną zmienną o nazwie `cheese`. W przypadku `self.cheese = 'Franek'` jest oczywiste, że masz na myśli atrybut instancji `self.cheese`.

Uczymy się mówić obiektowo

W tym ćwiczeniu nauczę Cię mówić „obektowo”. Podam niewielki zestaw słów z definicjami, które musisz znać. Potem wypiszę zestaw zdań z pustymi miejscami, które będziesz musiał zrozumieć. Na koniec otrzymasz duży zestaw ćwiczeń, które musisz wykonać, aby te zdania utrwaliły się w Twoim słowniku.

Ćwiczmy słówka

class (klasa) — informacja dla Pythona, aby utworzył nowy typ rzeczy.

Obiekt — ma dwa znaczenia: najbardziej podstawowy typ rzeczy i każda instancja jakiejś rzeczy.

Instancja — to, co otrzymujesz, gdy powiesz Pythonowi, aby utworzył klasę.

def — sposób definiowania funkcji wewnątrz klasy.

self — wewnątrz funkcji w klasie **self** jest zmienną dla instancji (obiektu), do której uzyskujemy dostęp.

Dziedziczenie — koncepcja, że jedna klasa może dziedziczyć cechy po innej klasie, podobnie jak Ty możesz dziedziczyć po rodzicach.

Kompozycja — koncepcja, że klasa może być skomponowana (składać się) z innych klas jako części, podobnie jak samochód ma koła.

Atrybut — właściwość klasy wynikająca z kompozycji i zazwyczaj będąca zmienną.

Jest (ang. *is-a*) — wyrażenie, które mówi, że coś odziedziczy po czymś innym, na przykład „łosoś” *jest* „rybą”.

Ma (ang. *has-a*) — wyrażenie, które mówi, że coś jest skomponowane z innych rzeczy lub ma jakąś cechę, na przykład „łosoś” *ma* „usta”.

Poświęć trochę czasu, aby przygotować fiszki dla tych pojęć i nauczyć się ich na pamięć. Tak jak zwykle, nie będzie to miało dla Ciebie zbyt dużego sensu dopóki, dopóty nie skończysz tego ćwiczenia, ale musisz najpierw poznać podstawowe słowa.

Ćwiczmy zdania

Oto przykładowa lista fragmentów kodu Pythona oraz odpowiadających im polskich zdań.

class X(Y) — „Utwórz klasę o nazwie X, która jest Y”.

class X(object): def __init__(self, J) — „Klasa X ma `__init__`, które przyjmuje parametry `self` i J”.

class X(object): def M(self, J) — „Klasa X ma funkcję o nazwie M, która przyjmuje parametry `self` i J”.

foo = X() — „Ustaw foo na instancję klasy X”.

foo.M(J) — „Z foo weź funkcję M i wywołaj ją z parametrami self, J”.

foo.K = Q — „Z foo weź atrybut K i ustaw go na Q”.

Za każdym razem, gdy widzisz w tych zdaniach litery X, Y, M, J, K, Q oraz zmienną foo, możesz potraktować to jak puste miejsca. Przykładowo mógłbym również napisać te zdania w następujący sposób.

1. „Utwórz klasę o nazwie ???, która jest Y”.
2. „Klasa ??? ma `__init__`, które przyjmuje parametry self i ???”.
3. „Klasa ??? ma funkcję o nazwie ???, która przyjmuje parametry self i ???”.
4. „Ustaw ??? na instancję klasy ???”.
5. „Z ??? weź funkcję ??? i wywołaj ją z parametrami self, ???”.
6. „Z ??? weź atrybut ??? i ustaw go na ???”.

Tak jak poprzednio, zapisz te rzeczy na fiszkach i poćwicz z nimi. Fragment kodu Pythona umieść na przedniej stronie karteczki, a pełne zdanie na odwrocie. *Musisz* nauczyć się zawsze wypowiadać zdanie dokładnie tak samo za każdym razem, gdy widzisz daną formę kodu. Nie mniej więcej tak samo, ale dokładnie tak samo.

Ćwiczenie łączone

Ostatecznym przygotowaniem jest połączenie ćwiczenia słówek i zdań. W tym ćwiczeniu wykonaj następujące czynności.

1. Wylosuj fiszkę z fragmentami kodu i najpierw spróbuj zapamiętać kod.
2. Odwróć fiszkę i przeczytaj zdanie, a dla każdego słowa w zdaniu, które należy do naszego zestawu słówek, wylosuj fiszkę.
3. Naucz się na pamięć słówek z tego zdania.
4. Kontynuuj, aż się nie znudzisz, potem zrób sobie przerwę i znów poćwicz.

Test z czytania

Mam dla Ciebie niewielki skrypt Pythona, za pomocą którego będziesz mógł w nieskończoność ćwiczyć poznane słowa. Jest to prosty skrypt i nie powinien mieć większych problemów z jego zrozumieniem, a jedyną rzeczą, którą robi, jest użycie biblioteki o nazwie `urllib` do pobrania przygotowanej przeze mnie listy słów. Oto skrypt, który należy wpisać w pliku `oop_test.py`.

ex41.py

```
1 import random
2 from urllib.request import urlopen
3 import sys
```

```

4
5 WORD_URL = "http://learncodethehardway.org/words.txt"
6 WORDS = []
7
8 PHRASES = {
9     "class %%(%%)":":
10     "Utwórz klasę o nazwie %%, która jest %%.",
11     "class %(object):\n\tdef __init__(self, ***)" :
12     "Klasa %% ma __init__, które przyjmuje parametry self i ***.",
13     "class %(object):\n\tdef ***(self, @@@)":
14     "Klasa %% ma funkcję ***, która przyjmuje parametry self i @@@.",
15     "*** = %%%()":
16     "Ustaw *** na instancję klasy %%. ",
17     "***.***(@@@)":
18     "Z *** weź funkcję *** i wywołaj ją z parametrami self, @@@.",
19     "***.*** = '***'":
20     "Z *** weź atrybut *** i ustaw go na '***'."
21 }
22
23 # czy chcemy najpierw poćwiczyć wyrażenia
24 if len(sys.argv) == 2 and sys.argv[1] == "polski":
25     PHRASE_FIRST = True
26 else:
27     PHRASE_FIRST = False
28
29 # ładujemy słowa ze strony internetowej
30 for word in urlopen(WORD_URL).readlines():
31     WORDS.append(str(word.strip(), encoding="utf-8"))
32
33
34 def convert(snippet, phrase):
35     class_names = [w.capitalize() for w in
36                     random.sample(WORDS, snippet.count("%%"))]
37     other_names = random.sample(WORDS, snippet.count("***"))
38     results = []
39     param_names = []
40
41     for i in range(0, snippet.count("@@")):
42         param_count = random.randint(1,3)
43         param_names.append(', '.join(
44             random.sample(WORDS, param_count)))
45
46     for sentence in snippet, phrase:
47         result = sentence[:]
48
49         # podstawiamy nazwy klas
50         for word in class_names:
51             result = result.replace("%%", word, 1)
52
53         # podstawiamy pozostałe nazwy
54         for word in other_names:
55             result = result.replace("***", word, 1)
56

```



```

57         # podstawiamy listy parametrów
58         for word in param_names:
59             result = result.replace("@@@", word, 1)
60
61         results.append(result)
62
63     return results
64
65
66     # kontynuujemy, aż użytkownik nie wciśnie Ctrl+D (lub Ctrl+Z w systemie Windows)
67     try:
68         while True:
69             snippets = list(PHRASES.keys())
70             random.shuffle(snippets)
71
72             for snippet in snippets:
73                 phrase = PHRASES[snippet]
74                 question, answer = convert(snippet, phrase)
75                 if PHRASE_FIRST:
76                     question, answer = answer, question
77
78                 print(question)
79
80                 input("> ")
81                 print(f"ODPOWIEDŹ: {answer}\n\n")
82     except EOFError:
83         print("\nDo widzenia")

```

Uruchom powyższy skrypt i spróbuj przetłumaczyć te „wyrażenia obiektowe” na język polski. Powinieneś zauważyć, że słownik PHRASES ma obie formy i musisz po prostu wpisać właściwą.

Tłumaczenie ze zdań na kod

Następnie powinieneś uruchomić skrypt z opcją polski, żeby poćwiczyć tłumaczenie w drugą stronę.

```
$ python oop_test.py polski
```

Pamiętaj, że w tych wyrażeniach użyto nonsensownych słów. Częścią nauki czytania kodu jest zaprzestanie przykładania tak dużego znaczenia do nazw używanych dla zmiennych i klas. Często może Ci się trafić słowo, takie jak „Cork” (korek) i nagle stracisz wątek, ponieważ to słowo wprawia Cię w zakłopotanie. W powyższym przykładzie „Cork” jest po prostu dowolną nazwą wybraną dla klasy. Nie przykładaj do niego żadnego innego znaczenia, ale potraktuj je zgodnie ze wzorcami, które Ci pokazałem.

Poczytaj jeszcze więcej kodu

Mam teraz dla Ciebie nową misję, żebyś poczytał jeszcze więcej kodu i poszukał w nim wyrażeń, których właśnie się nauczyłeś. Znajdź pliki z klasami, a następnie wykonaj następujące czynności.

1. Zapisz nazwę każdej klasy i podaj inne klasy, po których ona dziedziczy.
2. Pod tą nazwą wypisz wszystkie funkcje, które posiada ta klasa, oraz parametry przyjmowane przez funkcje.
3. Wymień wszystkie atrybuty, których używa na swoim `self`.
4. Dla każdego atrybutu podaj klasę, którą jest ten atrybut.

Celem tego ćwiczenia jest to, żebyś poczytał prawdziwy kod i zaczął uczyć się „dopasowywania do wzorców”, porównując poznane wyrażenia z tym, jak są używane. Jeśli będziesz ćwiczyć wystarczająco długo, powinieneś zacząć dostrzegać wzorce „wołające” do Ciebie z kodu, podczas gdy przedtem wydawały się mglistymi pustymi miejscami, których nie znałeś.

Typowe pytania

Co robi `result = sentence[:]`? W Pythonie jest to sposób na kopiowanie listy. Wykorzystuje składnię wycinka listy `[:]`, aby w efekcie utworzyć wycinek od pierwszego do ostatniego elementu.

Ten skrypt trudno uruchomić! Na tym etapie powinieneś już być w stanie wpisać ten kod i go uruchomić. Zawiera on kilka drobnych sztuczek, ale nie ma w tym nic skomplikowanego. Po prostu wykonaj wszystkie czynności, których do tej pory nauczyłeś się o debugowaniu skryptów. Wpisz każdą linię, potwierdź, że jest **dokładnie** taka sama jak moja, i poszukaj w internecie wszystkiego, czego nie znasz.

To wciąż jest zbyt trudne! Możesz to zrobić. Nie spiesz się i jeśli musisz, wpiszuj znak po znaku, ale przepisz wszystko dokładnie i postaraj się zrozumieć, co robi każdy fragment kodu.

Jest, ma, obiekty i klasy

Ważną koncepcją, którą musisz zrozumieć, jest różnica między klasą a obiektem. Problem polega na tym, że nie ma żadnej namacalnej „różnicy” między klasą a obiektem. W rzeczywistości są to te same rzeczy w różnych punktach czasu. Zademonstruję to za pomocą koanu Zen.

Jaka jest różnica między rybą a łososiem?

Czy to pytanie trochę Cię skonfundowało? Naprawdę usiądź i pomyśl o tym przez minutę. To znaczy, że ryba i łosoś różnią się od siebie, ale chwileczkę, są tym samym, prawda? Łosoś jest *rodzajem* ryby, więc uważam, że nie różnią się od siebie. Jednak jednocześnie łosoś jest konkretnym *rodzajem* ryby, więc w rzeczywistości różni się od wszystkich pozostałych ryb. Właśnie dlatego jest łososiem, a nie halibutem. Więc łosoś i ryba są takie same, ale różne. Dziwne.

To pytanie jest dezorientujące, ponieważ większość ludzi nie myśli o prawdziwych rzeczach w ten sposób, ale rozumie je intuicyjnie. Nie musisz myśleć o różnicy między rybą a łososiem, ponieważ *wiesz*, w jaki sposób te dwie rzeczy są ze sobą powiązane. Wiesz bez konieczności rozumienia tego, że łosoś jest *rodzajem* ryby i że istnieją inne rodzaje ryb.

Pójdźmy krok dalej. Powiedzmy, że masz wiadro z trzema łososiami, a ponieważ jesteś miłą osobą, zdecydowałeś się nazwać je Franek, Jaś i Marysia. Teraz zastanów się nad kolejnym pytaniem.

Jaka jest różnica między Marysią a łososiem?

To pytanie również jest dziwne, ale jest nieco łatwiejsze niż pytanie o rybę i łososa. Wiesz, że Marysia jest łososiem, więc naprawdę się nie różni. Jest po prostu określoną „instancją” łososa. Jaś i Franek również są instancjami łososa. Co mam na myśli, gdy mówię instancja? Myślę, że zostali stworzeni z jakiegoś innego łososa, a teraz reprezentują coś rzeczywistego, co ma atrybuty łososiopodobne.

A teraz gimnastyka umysłu: Ryba jest klasą i Łosoś jest klasą, a Marysia jest obiektem. Pomyśl o tym przez chwilę. Przeanalizujmy to powoli i zobaczymy, czy załapiesz.

Ryba jest klasą, co znaczy, że nie jest czymś *rzeczywistym*, ale raczej słowem, które dołączamy do instancji rzeczy o podobnych atrybutach. Ma płetwy? Ma skrzela? Żyje w wodzie? W porządku, to pewnie ryba.

Nagle pojawia się ktoś z dyplomem i mówi: „Nie, mój młody przyjacielu, ta ryba to w rzeczywistości *Salmo Salar*, czule nazywany łososiem”. Ten uczony właśnie wyjaśnił lepiej pojęcie ryby i utworzył nową klasę o nazwie „Łosoś”, która ma bardziej konkretne atrybuty. Ma dłuższy nos, czerwone mięso, jest duży, zamieszkuje oceany lub słodkie wody, jest smaczny? Prawdopodobnie łosoś.

W końcu przychodzi kucharz i zwraca się do doktora: „Nie. Widzisz tego łososia? Nazywam go Marysią i zrobię z niej smaczny fileć z doskonałym sosem”. Teraz ta *instancja* łososia (który jest także instancją ryby) o imieniu Marysia zamieniła się w coś prawdziwego, co wypełnia brzuch. Stała się obiektem.

I wszystko jasne: Marysia jest rodzajem łososia, który jest rodzajem ryby — obiekt jest klasą, która jest inną klasą.

Jak to wygląda w kodzie

Jest to dziwna koncepcja, ale szczerze mówiąc, musisz przejmować się nią tylko wtedy, kiedy tworzysz nowe klasy lub używasz klasy. Pokażę Ci dwie sztuczki, które pomogą zorientować się, czy coś jest klasą, czy obiektem.

Najpierw musisz nauczyć się dwóch słów: „jest” (ang. *is-a*) i „ma” (ang. *has-a*). Zwrotu „jest” używamy, kiedy mówimy o obiektach i klasach powiązanych przez relacje klasowe. Zwrotu „ma” używamy, kiedy mówimy o obiektach i klasach, które są powiązane tylko dlatego, że się do siebie *odwołują*.

Teraz przeanalizuj poniższy fragment kodu i zastąp każde wystąpienie `## ??` komentarzem, który mówi, czy następna linia reprezentuje relację „jest”, czy „ma” i jaka jest ta relacja. Podałem w kodzie kilka przykładów, więc musisz po prostu napisać pozostałe.

Pamiętaj, że „jest” to relacja między rybą a łososiem, a „ma” to relacja pomiędzy łososiem i skrzelami.

ex42.py

```
1  ## Animal jest obiektem (tak, to trochę dezorientujące); wyjaśni to ćwiczenie
   dodatkowe w „Zrób to sam”
2  class Animal(object):
3      pass
4
5  ## ??
6  class Dog(Animal):
7
8      def __init__(self, name):
9          ## ??
10         self.name = name
11
12  ## ??
13  class Cat(Animal):
14
15      def __init__(self, name):
16          ## ??
17         self.name = name
18
19  ## ??
20  class Person(object):
21
```

```
22     def __init__(self, name):
23         ## ??
24         self.name = name
25
26         ## Person (osoba) ma pet (jakiegoś zwierząka)
27         self.pet = None
28
29     ## ??
30     class Employee(Person):
31
32         def __init__(self, name, salary):
33             ## ?? hmm, co to za dziwna magia?
34             super(Employee, self).__init__(name)
35             ## ??
36             self.salary = salary
37
38     ## ??
39     class Fish(object):
40         pass
41
42     ## ??
43     class Salmon(Fish):
44         pass
45
46     ## ??
47     class Halibut(Fish):
48         pass
49
50
51     ## rover jest Dog (psem)
52     rover = Dog("Rover")
53
54     ## ??
55     satan = Cat("Satan")
56
57     ## ??
58     mary = Person("Mary")
59
60     ## ??
61     mary.pet = satan
62
63     ## ??
64     frank = Employee("Frank", 120000)
65
66     ## ??
67     frank.pet = rover
68
69     ## ??
70     flipper = Fish()
71
72     ## ??
73     crouse = Salmon()
```

```
74
75  ## ??
76  harry = Halibut()
```

Na temat class Nazwa(object)

W Pythonie 3 nie trzeba dodawać (object) po nazwie klasy, ale społeczność Pythona uważa, że „wyraźnie jest lepiej niż domyślnie”, więc ja i inni eksperci Pythona zdecydowaliśmy się to dołączać. Możesz spotkać kod, w którym nie ma (object) po prostych klasach, a te klasy są całkowicie prawidłowe i będą działać z klasami, które mają (object). W tym momencie jest to po prostu dodatkowa dokumentacja i nie ma wpływu na działanie klas.

W Pythonie 2 istniała różnica pomiędzy tymi dwoma typami klas, ale teraz już nie musisz się tym przejmować. Jedyną trudnością związaną z używaniem (object) jest mentalna gimnastyka, gdy mówisz „class *Nazwa* to klasa typu object”. Może Ci się to teraz wydawać niejasne, ponieważ jest to klasa, która jest obiektem *Nazwa*, który jest klasą, ale nie przejmuj się tym. Po prostu pomyśl sobie, że class *Nazwa*(object) to „podstawowa prosta klasa”, i wszystko będzie dobrze.

Oczywiście w przyszłości style i gusty programistów Pythona mogą ulec zmianie i to bezpośrednie użycie (object) może być postrzegane jako znak, że jesteś złym programistą. Jeśli tak się stanie, po prostu przestań tego używać lub powiedz im: „Zen Pythona mówi, że wyraźnie jest lepiej niż domyślnie”.

Zrób to sam

1. Sprawdź, dlaczego Python dodał tę dziwną klasę object i co to znaczy.
2. Czy możliwe jest użycie klasy tak, jakby była obiektem?
3. Do klas z tego ćwiczenia dopisz funkcje, które pozwalają robić im różne rzeczy. Zobacz, co się dzieje, gdy funkcje znajdują się w „klasie bazowej”, na przykład kiedy Animal ma funkcję w klasie bazowej Dog.
4. Poszukaj cudzych kodów i rozpracuj wszystkie relacje „jest” i „ma”.
5. Utwórz kilka nowych relacji, które są listami i słownikami, żebyś mógł również wprowadzić relacje „ma wiele”.
6. Czy uważasz, że istnieje coś takiego jak relacja „jest wiele”? Poczytaj o „wielokrotnym dziedziczeniu”, a następnie unikaj go, jeśli tylko możesz.

Typowe pytania

Po co komentarze `## ???` Są to komentarze typu „wypełnij puste miejsca”, które należy zastąpić opisami odpowiednich relacji „jest” i „ma”. Przeczytaj to ćwiczenie ponownie i przyjrzyj się innym komentarzom, aby zrozumieć, co mam na myśli.

Jaki jest sens `self.pet = None`? To gwarantuje, że atrybut `self.pet` tej klasy jest ustawiony na domyślną wartość `None` (brak wartości).

Co robi `super(Employee, self).__init__(name)`? W ten sposób możesz niezawodnie uruchomić metodę `__init__` klasy nadrzędnej. Wyszukaj w internecie „python3.6 super” i poczytaj różne rady dotyczące tego tematu.

Podstawowa analiza obiektowa i projekt

O piszę teraz proces używany do budowania czegoś za pomocą Pythona, szczególnie w programowaniu obiektowym (OOP). Przez „proces” mam na myśli zestaw czynności, które należy wykonywać po kolei, ale nie powinienem czuć się niewolnikiem tych procedur i nie zawsze będą one sprawdzać się dla każdego problemu. Są po prostu dobrym punktem wyjścia dla wielu problemów programistycznych, ale nie należy ich uważać za *jedyny* sposób rozwiązywania tego typu problemów. Ten proces to tylko jeden ze sposobów, z których można skorzystać.

Oto ten proces.

1. Opisz lub rozrysuj problem.
2. Wyodrębnij kluczowe pojęcia z punktu 1. i zbadaj je.
3. Utwórz hierarchię klas i mapę obiektów dla tych koncepcji.
4. Zakoduj klasy i napisz test, aby je uruchomić.
5. Powtórz i udoskonal.

Jest to tak zwane podejście „z góry na dół” (ang. *top down*) , co oznacza, że zaczynamy od bardzo abstrakcyjnego luźnego pomysłu, a następnie powoli udoskonalamy go, aż nabierze solidnych kształtów i będzie nadawał się do zakodowania.

Zaczynam od opisu problemu i próbuję zebrać wszystkie pomysły, które przychodzą mi do głowy na ten temat. Może nawet narysuję jakiś diagram lub mapę albo napiszę sam do siebie kilka e-maili opisujących problem. Pozwala mi to wyrazić kluczowe koncepcje problemu, a także zbadać własną wiedzę w danym zakresie.

Następnie przeglądam te notatki, rysunki oraz opisy i wyodrębniam kluczowe pojęcia. Stosuję przy tym jedną prostą sztuczkę: na podstawie moich notatek i rysunków sporządzam listę wszystkich *rzeczowników* i *czasowników*, a następnie opisuję ich wzajemne powiązania. W ten sposób otrzymuję porządną listę nazw klas, obiektów i funkcji, będącą podstawą następnego kroku. Biorę tę listę koncepcji, a następnie wyszukuję wszystko, czego nie rozumiem, żebym mógł je udoskonalić, jeśli będzie trzeba.

Kiedy mam listę koncepcji, tworzę ich prosty zarys (drzewo) i opisuję relacje klas. Zazwyczaj, posługując się listą rzeczowników, zadaję sobie pytanie: „Czy ten rzeczownik przypomina inne rzeczowniki-koncepcje? To znaczy, że mają wspólną klasę nadrzędną, więc jak będzie się ona nazywać?” Robię to, dopóki nie otrzymam hierarchii klas, która jest prostą listą o strukturze drzewa lub diagramem. Potem zabieram się za spisane *czasowniki* i sprawdzam, czy są to nazwy funkcji dla każdej klasy, następnie umieszczam je w moim drzewie.

Po ustaleniu hierarchii klas siadam i piszę podstawowy szkielet kodu, który zawiera jedynie klasy, ich funkcje i nic więcej. Następnie piszę test, który uruchamia ten kod, i upewniam się,

że utworzone przeze mnie klasy mają sens i działają poprawnie. Czasami piszę ten test najpierw, a innym razem piszę trochę testu, trochę kodu, znów trochę testu i tak dalej, aż nie zbuduję całości.

Na koniec powtarzam cały proces, udoskonalając po kolei poszczególne elementy, aby całość była możliwie jasna i przejrzysta, gdy przystąpię do kolejnych implementacji. Jeśli utknę w jakiejś określonej części z powodu koncepcji lub problemu, których nie przewidziałem, wtedy siadam i rozpoczynam od nowa proces tylko dla tej części, aby dowiedzieć się więcej, zanim przejdę dalej.

Przejdę teraz przez ten proces, wymyślając silnik gry i grę do tego ćwiczenia.

Analiza prostego silnika gry

Gra, którą chcę opracować, nazywa się *Goci z planety Percal 25*. Będzie to niewielka kosmiczna gra przygodowa. Nie mając w głowie nic poza czystą koncepcją, mogę zgłębić ten pomysł i wykombinować, jak powołać tę grę do życia.

Opisz lub rozrysuj problem

Napiszę krótki akapit z zarysem gry.

„Obcy zaatakowali statek kosmiczny, a nasz bohater musi pokonać labirynt pokoi, walcząc z przeciwnikiem, aby dostać się do kapsuły ratunkowej i uciec na pobliską planetę. Ta gra będzie przypominać gry tekstowe, takie jak *Zork* lub *Adventure*, z ich zabawnymi sposobami uśmiercania bohatera. Gra będzie obejmować silnik, który uruchamia mapę pełną pokoi lub scen. Każdy pokój wydrukuje własny opis, gdy gracz do niego wejdzie, a następnie wskaże silnikowi, który pokój z mapy ma być uruchomiony jako następny”.

W tym momencie mam już dobry pomysł na grę i sposób jej działania, więc teraz opiszę każdą scenę.

Śmierć (ang. *death*) — gracz umiera i powinno to być zabawne.

Centralny korytarz (ang. *central corridor*) — punkt startowy gry. W korytarzu stoi już jeden Got, którego musisz pokonać żartem, zanim przejdiesz dalej.

Magazyn broni laserowej (ang. *laser weapon armory*) — tutaj bohater znajduje bombę neutronową do wysadzenia statku, którą musi podłożyć, zanim dostanie się do kapsuły ratunkowej. Bomba jest umieszczona w pojemniku zabezpieczonym wpisywanym na klawiaturze kodem numerycznym, który bohater musi odgadnąć.

Mostek (ang. *the bridge*) — kolejna bitwa z Gotami, podczas której bohater podkłada bombę.

Kapsuła ratunkowa (ang. *escape pod*) — scena, w której bohater ucieka, ale tylko pod warunkiem wybrania właściwej kapsuły ratunkowej.

Na tym etapie mógłbym sporządzić mapę scen albo więcej opisów każdego pokoju — cokolwiek przyjdzie mi do głowy w trakcie analizowania problemu.

Wyodrębnij kluczowe pojęcia i zbadaj je

Mam teraz wystarczająco dużo informacji, aby wyodrębnić niektóre rzeczowniki i przeanalizować ich hierarchię klas. Najpierw tworzę listę wszystkich rzeczowników:

- obcy (ang. *alien*),
- gracz (ang. *player*),
- statek (ang. *ship*),
- labirynt (ang. *maze*),
- pokój (ang. *room*),
- scena (ang. *scene*),
- Got (ang. *Gothon*),
- kapsuła ratunkowa,
- planeta (ang. *planet*),
- mapa (ang. *map*),
- silnik (ang. *engine*),
- śmierć,
- centralny korytarz,
- magazyn broni laserowej,
- mostek.

Mógłbym również przejrzeć wszystkie czasowniki i sprawdzić, czy któreś z nich mogą być dobrymi nazwami funkcji, ale na razie to pomiję.

Na tym etapie mogę również zbadać każdą z tych koncepcji i wszystko, o czym jeszcze nie wiem. Mogę na przykład zagrać w kilka takich gier i upewnić się, że wiem, jak działają. Mogę dowiedzieć się, w jaki sposób projektuje się statki lub jak działają bomby. Być może spróbuję rozwiązać pewne problemy techniczne, takie jak przechowywanie stanu gry w bazie danych. Po wykonaniu tych badań mogę zacząć od nowa w punkcie 1., wykorzystując nowo pozyskane informacje, by poprawić mój opis i wyodrębnić nowe koncepcje.

Utwórz hierarchię klas i mapę obiektów dla koncepcji

Po wykonaniu tych czynności tworzę hierarchię klas, zadając sobie pytanie: „Co jest podobne do innych rzeczy?”. Pytam też: „Co jest po prostu inną nazwą dla innej rzeczy?”.

Od razu widzę, że „pokój” i „scena” to w zasadzie to samo, w zależności od tego, jak chcę do tego podejść. Dla tej gry wybieram „scenę”. Następnie widzę, że wszystkie konkretne pokoje, takie jak „centralny korytarz”, to zasadniczo sceny. Zauważam też, że „śmierć” to w zasadzie scena, co potwierdza mój wybór sceny zamiast pokoju, ponieważ można mieć scenę śmierci, ale pokój śmierci jest trochę dziwny. „Labirynt” i „mapa” to — ogólnie rzecz biorąc — to samo, więc wybiorę mapę, ponieważ częściej jej używałem. Nie chcę tworzyć systemu walki, więc zignoruję „obcego” i „gracza” i odłożę to na później. „Planeta” mogłaby być po prostu kolejną sceną zamiast czymś konkretnym.

Po tym procesie myślowym zaczynam tworzyć hierarchię klas, która w moim edytorze tekstu wygląda tak.

```
* Map
* Engine
* Scene
  * Death
  * Central Corridor
  * Laser Weapon Armory
  * The Bridge
  * Escape Pod
```

Następnie analizuję ją i zastanawiam się, jakie działania są potrzebne dla każdej rzeczy według czasowników z opisu. Z opisu wiem na przykład, że będę potrzebował sposobu na „uruchamianie” silnika, „pobieranie kolejnej sceny” z mapy, pobieranie „sceny otwierającej” i „wchodzenie” w scenę. Dodam to w ten sposób.

```
* Map
  - next_scene (następna scena)
  - opening_scene (scena otwierająca)
* Engine
  - play (uruchamianie)
* Scene
  - enter (wchodzenie)
* Death
* Central Corridor
* Laser Weapon Armory
* The Bridge
* Escape Pod
```

Zwróć uwagę, że umieściłem -enter po prostu pod Scene, ponieważ wiem, że wszystkie sceny poniżej będą to dziedziczyć i trzeba to później nadpisać.

Zakoduj klasy i napisz test, aby je uruchomić

Kiedy mam już drzewo klas i niektóre funkcje, otwieram w edytorze plik źródłowy i próbuję napisać kod. Zwykle po prostu kopiuję i wklejam do pliku źródłowego drzewo, a następnie edytuję je, żeby powstały klasy. Oto mały przykład, jak może to wyglądać na początku, z prostym krótkim testem na końcu pliku.

ex43_classes.py

```
1  class Scene(object):
2
3      def enter(self):
4          pass
5
6
7  class Engine(object):
8
9      def __init__(self, scene_map):
10         pass
11
```

```
12         def play(self):
13             pass
14
15     class Death(Scene):
16
17         def enter(self):
18             pass
19
20     class CentralCorridor(Scene):
21
22         def enter(self):
23             pass
24
25     class LaserWeaponArmory(Scene):
26
27         def enter(self):
28             pass
29
30     class TheBridge(Scene):
31
32         def enter(self):
33             pass
34
35     class EscapePod(Scene):
36
37         def enter(self):
38             pass
39
40
41     class Map(object):
42
43         def __init__(self, start_scene):
44             pass
45
46         def next_scene(self, scene_name):
47             pass
48
49         def opening_scene(self):
50             pass
51
52
53     a_map = Map('central_corridor')
54     a_game = Engine(a_map)
55     a_game.play()
```

W tym pliku możesz zobaczyć, że zreplikowałem po prostu żadaną hierarchię, a następnie dodałem trochę kodu na końcu, aby uruchomić tę podstawową strukturę i sprawdzić, czy wszystko działa. W kolejnych częściach tego ćwiczenia wypełnisz resztę tego kodu i zmodyfikujesz go tak, aby pasował do opisu gry.

Powtórz i udoskonal

Ostatni krok w moim małym procesie to nie tyle krok, co pętla *while*. Nigdy nie robi się tego wszystkiego jako operacji jednorazowej. Po prostu wracasz do początku, powtarzasz cały proces i udoskonalasz go na podstawie informacji, które pozyskałeś w poprzednich krokach. Czasami dochodzę do punktu 3. i zdaję sobie sprawę, że muszę jeszcze popracować nad punktami 1. i 2., więc zatrzymuję się i wracam do nich. Czasami w przypływie natchnienia przeskakuję na koniec i koduję rozwiązanie, które mam w głowie, zanim gdzieś mi umknie, ale potem wracam do poprzednich kroków, aby upewnić się, że uwzględniłem wszystkie możliwości.

Można też potraktować ten proces nie jako coś, co robi się na jednym poziomie, ale można zrobić na każdym poziomie, gdy napotyka się konkretny problem. Powiedzmy, że nie wiem jeszcze, jak napisać metodę `Engine.play`. Mogę zatrzymać się i wykonać cały ten proces *tylko* na jednej funkcji, aby się dowiedzieć, jak ją napisać.

Z góry do dołu i z dołu do góry

Proces ten jest zazwyczaj określany jako podejście „z góry na dół”, ponieważ zaczyna się od najbardziej abstrakcyjnych pojęć (góry) i przechodzi stopniowo do rzeczywistej implementacji. Chcę, żebyś od tej chwili używał procesu, który właśnie opisałem, podczas analizowania problemów z tej książki. Powinieneś jednak wiedzieć, że istnieje również inny sposób rozwiązywania problemów w programowaniu, który zaczyna się od kodu i idzie w „górze” do abstrakcyjnych pojęć. Ten inny sposób nazywa się podejściem „z dołu do góry” (ang. *bottom up*). Oto ogólne kroki wykonywane w tym podejściu.

1. Weź niewielki problem, napisz trochę kodu i spraw, żeby jako tako działał.
2. Udoskonal kod do bardziej formalnej postaci z klasami i zautomatyzowanymi testami.
3. Wyodrębnij kluczowe pojęcia, których używasz, i zbadaj je.
4. Opisz to, co się naprawdę dzieje się w kodzie.
5. Wróć i udoskonal kod, prawdopodobnie wyrzucając go do kosza i zaczynając od nowa.
6. Powtórz czynności, przechodząc do innej części problemu.

Uważam, że ten proces jest lepszy, gdy jesteś już bardziej zaawansowanym programistą i naturalnie myślisz o problemach w kategoriach kodu. Ten proces jest bardzo dobry, gdy znasz niewielkie kawałki ogólnej układanki, ale być może nie masz jeszcze wystarczająco dużo informacji na temat ogólnej koncepcji. Wtedy rozbicie problemu na małe kawałki i badanie go za pomocą kodu pomaga powoli pracować nad nim dopóki, dopóty go nie rozwiążesz. Pamiętaj jednak, że Twoje rozwiązanie prawdopodobnie będzie chaotyczne i dziwne, dlatego moja wersja tego procesu wymaga wracania do początku i przeprowadzania badań, a następnie czyszczenia rozwiązania na podstawie tego, czego się nauczyłeś.

Kod gry Goci z planety Percal 25

Stop! Pokażę Ci moje ostateczne rozwiązanie powyższego problemu, ale nie chcę, żebyś po prostu usiadł i wpisał ten kod. Masz wziąć ogólny szkielet kodu, który przygotowałem, i *samodzielnie* spróbować na podstawie opisu zrobić z niego działający skrypt. Kiedy będziesz miał już rozwiązanie, możesz wrócić i zobaczyć, jak to zrobiłem.

Zamiast wrzucać na raz cały kod, rozbiję ten finalny plik `ex43.py` na sekcje i objaśnię po kolei każdą z nich.

`ex43.py`

```
1  from sys import exit
2  from random import randint
3  from textwrap import dedent
```

To są po prostu podstawowe importy dla gry. Jediną nową rzeczą jest zaimportowanie funkcji `dedent` z modułu `textwrap`. Ta mała funkcja pomoże napisać nasze opisy pokoi za pomocą łańcuchów znaków w potrójnych cudzysłowach (`"""`). Usuwa ona po prostu znaki niedrukowalne z początków linii w łańcuchu znaków. Bez tej funkcji nie dałoby się użyć łańcuchów znaków w potrójnych cudzysłowach, ponieważ *na ekranie* byłyby one wcięte o tyle samo spacji, co w kodzie *Pythona*.

`ex43.py`

```
1  class Scene(object):
2
3      def enter(self):
4          print("Ta scena nie jest jeszcze skonfigurowana.")
5          print("Utwórz jej podklasę i zaimplementuj enter().")
6          exit(1)
```

Jak widzisz w szkieletowym kodzie, mam klasę bazową dla `Scene`, która będzie miała wspólne rzeczy, jakie robią wszystkie sceny. W tym prostym programie nie robią zbyt wiele, więc jest to raczej demonstracja tego, co należy zrobić, aby utworzyć klasę bazową.

`ex43.py`

```
1  class Engine(object):
2
3      def __init__(self, scene_map):
4          self.scene_map = scene_map
5
6      def play(self):
7          current_scene = self.scene_map.opening_scene()
8          last_scene = self.scene_map.next_scene('finished')
9
10         while current_scene != last_scene:
11             next_scene_name = current_scene.enter()
12             current_scene = self.scene_map.next_scene(next_scene_name)
```



```

13
14         # pamiętaj o wydrukowaniu ostatniej sceny
15         current_scene.enter()

```

Mam również klasę `Engine` i możesz zauważyć, w jaki sposób używam już metod dla `Map.opening_scene` i `Map.next_scene`. Ponieważ przygotowałem sobie pewien plan, mogę po prostu założyć, że napiszę je, a następnie ich użyję, zanim napiszę klasę `Map`.

ex43.py

```

1  class Death(Scene):
2
3      quips = [
4          "Umarłeś. Jesteś w tym do bani.",
5          "Twoja matka byłaby dumna... gdyby była mądrzejsza.",
6          "Ale z Ciebie ciołek.",
7          "Mam szczeniaka, który robi to lepiej.",
8          "Jesteś gorszy niż dowcipy Twojego ojca."
9
10     ]
11
12     def enter(self):
13         print(Death.quips[randint(0, len(self.quips)-1)])
14         exit(1)

```

Jako pierwszą napisałem tę dziwną scenę o nazwie `Death`, która pokazuje najprostszy rodzaj sceny, jaką możesz napisać.

ex43.py

```

1  class CentralCorridor(Scene):
2
3      def enter(self):
4          print(dedent("""
5              Goci z planety Percal 25 wdarli się na Twój statek i
6              wymordowali całą załogę. Jesteś ostatnim ocalałym członkiem
7              załogi, a Twoją ostatnią misją jest zdobycie bomby neutronowej
8              z magazynu broni, umieszczenie jej na mostku i wysadzenie
9              statku, gdy już uda Ci się dotrzeć do kapsuły ratunkowej.
10
11              Biegniesz centralnym korytarzem w kierunku magazynu broni,
12              gdy nagle wyskakuje jakiś Got o czerwonej, łuszczącej się skórze,
13              z czarnymi zębami, ubrany w kostium złego klauna i cały ziejący
14              nienawiścią. Blokuje dostęp do magazynu broni i właśnie zamierza
15              wyciągnąć broń, żeby Cię rozwalić.
16              """))
17
18          action = input("> ")
19
20          if action == "strzelam!":
21              print(dedent("""
22                  Błyskawicznie dobywasz z kabury miotacz i strzelasz
23                  do Gota. Kostium klauna powiewa i wije się wokół jego

```

```

24         ciała, przez co nie możesz dobrze wycelować. Promień
25         lasera trafia w jego kostium, ale całkowicie omija ciało.
26         Zupełnie nowy kostium, który kupiła mu mama, jest kompletnie
27         zniszczony, co wywołuje w nim wulkan wściekłości i
28         strzela Ci wielokrotnie w głowę, aż padasz martwy.
29         Wtedy Cię zjada.
30         """))
31     return 'death'
32
33     elif action == "robię unik!":
34         print(dedent("""
35             Jak światowej klasy bokser robisz unik, przekręcasz się
36             i prześlizgujesz w prawo, gdy miotacz Gota tnie laserem
37             obok Twojej głowy. W trakcie wykonywania tego artystycznego
38             uniku potykasz się o własną stopę, walisz głową w metalową
39             ścianę i tracisz przytomność. Po chwili budzisz się tylko
40             po to, żeby umrzeć stratowany i zjedzony przez Gota.
41             """))
42     return 'death'
43
44     elif action == "opowiadam dowcip":
45         print(dedent("""
46             Na Twoje szczęście w akademii nauczyli Cię rzucać mięsem
47             w obcych językach. Opowiadasz jedyny gocki żart, jaki znasz:
48             Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr,
49             fur fvgf nebhaq gur ubhfr. Got zastyga, przez chwilę próbuje
50             się powstrzymać, a następnie wybucha śmiechem i nie może
51             się
52             ruszyć. W tym czasie szybko uciekasz i na odchodnym strzelasz
53             mu prosto w głowę, powalając go trupem, a następnie znikasz
54             za drzwiami magazynu broni.
55             """))
56         return 'laser_weapon_armory'
57     else:
58         print("NIE MOŻNA PRZELICZYĆ!")
59         return 'central_corridor'

```

Potem utworzyłem scenę CentralCorridor, która jest początkiem gry. Piszę sceny dla gry przed utworzeniem klasy Map, ponieważ muszę odwoływać się do nich później. Zwróć również uwagę na sposób, w jaki użyłem funkcji dedent w linii 4. Spróbuj później ją usunąć, aby zobaczyć, co robi.

ex43.py

```

1  class LaserWeaponArmory(Scene):
2
3      def enter(self):
4          print(dedent("""
5              Dajesz nura do magazynu broni, przykucasz i lustrujesz wzrokiem
6              pomieszczenie, szukając ukrywających się Gotów. Panuje martwa,
7              przerażająca cisza. Wstajesz i biegniesz w odległy koniec
8              pomieszczenia, gdzie znajdujesz bombę neutronową umieszczoną w

```

```

9         zabezpieczonym pojemniku. Aby wyjąć bombę, musisz odblokować
10        zamek, wpisując na klawiaturze kod. Jeśli 10 razy wpiszesz
11        niewłaściwy kod, zamek zablokuje się na zawsze i nie wyjmiesz
12        bomby. Kod ma 3 cyfry.
13        """))
14
15        code = f"{randint(1,9)}{randint(1,9)}{randint(1,9)}"
16        guess = input("[keypad]> ")
17        guesses = 0
18
19        while guess != code and guesses < 10:
20            print("BZZZZEDDD!")
21            guesses += 1
22            guess = input("[keypad]> ")
23
24        if guess == code:
25            print(dedent("""
26                Blokada puszcza i pojemnik otwiera się, z sykiem wypuszczając
27                ze środka gaz. Chwytasz bombę neutronową i biegniesz tak
28                szybko, jak możesz, w kierunku mostka, gdzie musisz podłożyć
29                ją w odpowiednim miejscu.
30                """))
31            return 'the_bridge'
32        else:
33            print(dedent("""
34                Brzęczyk blokady wybrzmiewa po raz ostatni i słyszysz
35                obrzydliwy dźwięk stapienia się mechanizmu blokady
36                pojemnika. Decydujesz się już nigdzie nie uciekać, a Goci
37                ze swojego statku wysadzają w powietrze Twój. Umierasz.
38                """))
39            return 'death'
40
41
42
43    class TheBridge(Scene):
44
45        def enter(self):
46            print(dedent("""
47                Zdyszany wpadasz na mostek z bombą neutronową pod
48                pachą i zaskakujesz 5 Gotów, którzy próbują przejąć
49                kontrolę nad statkiem. Każdy kolejny ma jeszcze brzydszy
50                kostium klauna niż poprzedni. Żaden z nich nie wyciągnął
51                jeszcze broni, ponieważ widzą aktywowaną bombę pod Twoją
52                pachą i nie chcą jej przypadkowo zdetonować.
53                """))
54
55            action = input("> ")
56
57            if action == "rzucam bombę":
58                print(dedent("""
59                    W panice rzucasz bombą w kierunku grupy Gotów
60                    i rzucasz się do drzwi. Goci reagują natychmiastowo
61                    i strzelają Ci prosto w plecy, kładąc Cię trupem.

```

```

62         Konając, widzisz jeszcze Gota, który gorączkowo próbuje
63         rozbroić bombę. Umierasz ze świadomością, że prawdopodobnie
64         wszyscy wylecą w powietrze, gdy bomba wybuchnie.
65         """)
66     return 'death'
67
68     elif action == "powoli podkładam bombę":
69         print(dedent("""
70             Przykładasz lufę miotacza do trzymanej pod pachą bomby,
71             a Goci podnoszą ręce do góry i zaczynają się pocić.
72             Cofasz się powoli do drzwi, otwierasz je, a potem
73             ostrożnie kładziesz bombę na podłodze, nadal celując
74             w nią miotaczem. Następnie przeskakujesz przez drzwi,
75             zamykasz je przyciskiem i strzelasz w panel kontrolny
76             zamka, aby nie mogli się wydostać. Bomba podłożona, więc
77             uciekasz do kapsuły ratunkowej, aby wydostać się z tej puszki.
78             """))
79
80     return 'escape_pod'
81 else:
82     print("NIE MOŻNA PRZELICZYĆ!")
83     return "the_bridge"
84
85
86 class EscapePod(Scene):
87
88     def enter(self):
89         print(dedent("""
90             Biegniesz korytarzami, rozpaczliwie próbując dotrzeć
91             do kapsuły, zanim cały statek eksploduje. Wydaje się,
92             że na statku nie ma już prawie żadnego Gota, więc masz
93             wolną drogę ucieczki. Docierasz do komory z kapsułami
94             ratunkowymi i teraz musisz zdecydować się na jedną z nich.
95             Niektóre mogły zostać uszkodzone, ale nie masz czasu sprawdzać.
96             Jest 5 kapsuł, którą wybierasz?
97             """))
98
99         good_pod = randint(1,5)
100         guess = input("[pod #]> ")
101
102
103         if int(guess) != good_pod:
104             print(dedent("""
105                 Wskakujesz do kapsuły {guess} i katapultujesz się.
106                 Kapsuła wystrzeliwuje w otchłań kosmosu, a następnie
107                 imploduje, gdy kadłub pęka i miażdży Twoje ciało,
108                 robiąc z niego dżem malinowy.
109                 """))
110             return 'death'
111         else:
112             print(dedent("""
113                 Wskakujesz do kapsuły {guess} i katapultujesz się.
114                 Kapsuła lekko wślizguje się w kosmos, zmierzając w

```

```

115             kierunku najbliższej planety. W trakcie lotu spoglądasz
116             wstecz i widzisz, że Twój statek imploduje, a następnie
117             eksploduje, jak jasna gwiazda, niszcząc jednocześnie
118             statek Gotów. Wygrałeś!
119             """))
120
121         return 'finished'
122
123     class Finished(Scene):
124
125         def enter(self):
126             print("Wygrałeś! Dobra robota.")
127             return 'finished'

```

To jest reszta scen gry, a ponieważ wiem, że ich potrzebuję, i przemyślałem, w jaki sposób będą poukładane, mogę je zakodować bezpośrednio.

Nawiasem mówiąc, nie wpisałbym po prostu całego tego kodu. Pamiętaj, że powiedziałem, aby próbować i budować grę stopniowo, po kawałku. Po prostu pokazuję Ci rezultat.

ex43.py

```

1  class Map(object):
2
3      scenes = {
4          'central_corridor': CentralCorridor(),
5          'laser_weapon_armory': LaserWeaponArmory(),
6          'the_bridge': TheBridge(),
7          'escape_pod': EscapePod(),
8          'death': Death(),
9          'finished': Finished(),
10     }
11
12     def __init__(self, start_scene):
13         self.start_scene = start_scene
14
15     def next_scene(self, scene_name):
16         val = Map.scenes.get(scene_name)
17         return val
18
19     def opening_scene(self):
20         return self.next_scene(self.start_scene)

```

Następnie mam klasę Map — widać, że przechowuje ona w słowniku każdą scenę według nazwy. Odwołuję się potem do tego słownika za pomocą Map.scenes. Właśnie dlatego mapa następuje po scenach — słownik musi odnosić się do scen, więc muszą one istnieć.

ex43.py

```

1  a_map = Map('central_corridor')
2  a_game = Engine(a_map)
3  a_game.play()

```

Na koniec mam kod, który uruchamia grę, tworząc mapę (Map), a następnie przekazuje tę mapę do silnika (Engine) przed wywołaniem `play` w celu uruchomienia gry.

Co powinieneś zobaczyć

Upewnij się, że rozumiesz grę i koniecznie spróbuj najpierw napisać rozwiązanie samodzielnie. Jeśli coś naprawdę zabije Ci klina, możesz trochę pooszukiwać, zerkając do mojego kodu, ale potem kontynuuj próbę rozwiązania tego problemu na własną rękę.

Kiedy uruchamiam moją grę, wygląda to tak.

Ćwiczenie 43. sesja

```
$ python3.6 ex43.py
```

Goci z planety Percal 25 wdarli się na Twój statek i wymordowali całą załogę. Jesteś ostatnim ocalałym członkiem załogi, a Twoją ostatnią misją jest zdobycie bomby neutronowej z magazynu broni, umieszczenie jej na mostku i wysadzenie statku, gdy już uda Ci się dotrzeć do kapsuły ratunkowej.

Biegniesz centralnym korytarzem w kierunku magazynu broni, gdy nagle wyskakuje jakiś Got o czerwonej, łuszczącej się skórze, z czarnymi zębami, ubrany w kostium złego klauna i cały ziejący nienawiścią. Blokuję dostęp do magazynu broni i właśnie zamierza wyciągnąć broń, żeby Cię rozwalić.

> robię unik!

Jak światowej klasy bokser robisz unik, przekręcasz się i prześlizgujesz w prawo, gdy miotacz Gota tnie laserem obok Twojej głowy. W trakcie wykonywania tego artystycznego uniku potykasz się o własną stopę, walisz głową w metalową ścianę i tracisz przytomność. Po chwili budzisz się tylko po to, żeby umrzeć stratowany i zjedzony przez Gota.

Jesteś gorszy niż dowcipy Twojego ojca.

Zrób to sam

1. Zmień tę grę! Może jej nie cierpisz? Być może jest w niej zbyt dużo przemocy albo nie przepadasz za science-fiction? Spraw, aby gra działała, a następnie zmień ją, tak jak chcesz. To Twój komputer; każesz mu robić, co chcesz.
2. W tym kodzie mam błąd. Dlaczego kod blokady pojemnika z bombą zgaduje się 11 razy?
3. Wyjaśnij, jak działa zwracanie następnego pokoju.
4. Dodaj do gry kody przejścia, aby można było przechodzić przez trudniejsze pokoje. Ja mogę to zrobić za pomocą dwóch słów w jednej linii.

5. Wróć do mojego opisu i analizy, a następnie spróbuj zbudować niewielki system walki dla bohatera i różnych napotykanych przez niego Gotów.
6. Jest to właściwie niewielka wersja tak zwanego „automatu skończonego” (ang. *finite state machine*). Poczytaj o tych automatach. Może nie do końca to zrozumiesz, ale spróbuj.

Typowe pytania

Gdzie mogę znaleźć historie do własnych gier? Możesz je wymyślić, tak jak opowiadasz różne historyjki znajomym. Możesz też zaczerpnąć proste sceny z książki lub filmu, które Ci się podobają.

Porównanie dziedziczenia i kompozycji

W bajkach o bohaterach pokonujących straszliwych złoczyńców zawsze jest jakiś ciemny las. Może to być jaskinia, las, inna planeta lub po prostu jakieś miejsce, do którego — co każdy wie — bohater nie powinien się udawać. Oczywiście zaraz po tym, jak poznajemy postać złoczyńcy, dowiadujemy się, że bohater musi pójść do tego lasu, żeby zabić złego gościa. Wygląda na to, że nasz bohater po prostu ciągle pakuje się w sytuacje, które wymagają od niego narażania życia w tym ciemnym lesie.

Rzadko czytamy bajki o bohaterach, którzy są wystarczająco inteligentni, aby po prostu uniknąć tej sytuacji. Nigdy nie słyszy się, żeby jakiś bohater mówił: „Chwileczkę, jeśli wypłynę na pełne morze w poszukiwaniu szczęścia, zostawiając za sobą moją piękną księżniczkę, mogę umrzeć, a ona będzie musiała poślubić jakiegoś brzydkiego księcia o imieniu Humperdink. Humperdink! Myślę, że zostanę tutaj i założę agencję *Wieśniak do wynajęcia*”. Gdyby tak zrobił, nie byłoby ognistych bagien, umierania, reanimowania, walk na miecze, olbrzymów ani w ogóle żadnej opowieści. Z tego powodu lasy w tych historiach zdają się istnieć tylko po to, by jak czarna dziura wciągać bohatera, bez względu na to, co robi.

W programowaniu obiektowym dziedziczenie jest złym lasem. Doświadczeni programiści wiedzą, że należy unikać tego zła, ponieważ zdają sobie sprawę, że gdzieś w głębi Mrocznego Lasu Dziedziczenia mieszka Zła Królowa Wielokrotnego Dziedziczenia. Lubi swoimi ogromnymi zębiskami złożoności pożerać oprogramowanie i programistów, przeżuwać ścierwo upadłych. Jednak ten las jest tak potężny i kuszący, że prawie każdy programista musi do niego wkroczyć i spróbować wydostać się żywy z głową Złej Królowej, zanim będzie mógł nazywać siebie prawdziwym programistą. Nie możesz się po prostu oprzeć zewowi Lasu Dziedziczenia, więc wchodzisz do niego. Po tej przygodzie masz już nauczkę, żeby trzymać się z daleka od tego ciemnego lasu, a jeśli kiedykolwiek będziesz zmuszony wejść do niego ponownie, na pewno zabierzesz ze sobą całą armię.

W zabawny sposób chcę Ci powiedzieć, że będziesz się teraz uczył czegoś, czego używać powinienieś ostrożnie, a zwie się to **dziedziczeniem**. Programiści, którzy są obecnie w tym lesie i walczą z królową, prawdopodobnie powiedzą Ci, że musisz tam wejść. Mówią tak, ponieważ potrzebują Twojej pomocy, gdyż prawdopodobnie to, co stworzyli, przerasta nawet ich samych. Zawsze jednak powinieś pamiętać o jednej zasadzie. Oto ona.

Większość zastosowań dziedziczenia można uprościć lub zastąpić kompozycją, a wielokrotnego dziedziczenia należy unikać za wszelką cenę.

Co to jest dziedziczenie

Dziedziczenie służy do wskazania, że dana klasa uzyska większość swoich cech (lub wszystkie) z klasy nadrzędnej. Tak dzieje się domyślnie za każdym razem, gdy piszesz `class Foo(Bar)`, a to znaczy: „Utwórz klasę Foo, która dziedziczy z Bar”. Gdy tak robisz, język programowania

powoduje, że wszelkie akcje wykonywane na instancjach `Foo` działają również w taki sposób, jakby zostały wykonane na instancji `Bar`. Dzięki temu możesz umieścić ogólną funkcjonalność w klasie `Bar`, a następnie, jeśli trzeba, wyspecjalizować tę funkcjonalność w klasie `Foo`.

Gdy wprowadzasz tego rodzaju specjalizację, istnieją trzy sposoby interakcji klasy nadrzędnej, czyli rodzica (ang. *parent*), i klasy potomnej, czyli dziecka (ang. *child*).

1. Działania na dziecku implikują działanie na rodzicu.
2. Działania na dziecku nadpisują działanie na rodzicu.
3. Działania na dziecku zmieniają działanie na rodzicu.

Zademonstruję teraz każdy z tych sposobów po kolei i pokażę dla nich kod.

Dziedziczenie domyślne

Najpierw pokażę domyślne działania, które mają miejsce przy definiowaniu funkcji w klasie nadrzędnej, ale *nie* w klasie potomnej.

ex44a.py

```
1  class Parent(object):
2
3      def implicit(self):
4          print("RODZIC implicit()")
5
6  class Child(Parent):
7      pass
8
9  dad = Parent()
10 son = Child()
11
12 dad.implicit()
13 son.implicit()
```

Używając `pass` pod `class Child`, informujesz Pythona, że potrzebujesz pustego bloku. To tworzy klasę o nazwie `Child`, ale mówi, że nie ma w niej nic nowego do zdefiniowania. Zamiast tego odziedziczy ona wszystkie zachowania z klasy `Parent`. Po uruchomieniu tego kodu otrzymasz następujące dane wyjściowe.

Ćwiczenie 44a — sesja

```
$ python3.6 ex44a.py
RODZIC implicit()
RODZIC implicit()
```

Zwróć uwagę, że chociaż `Child` *nie ma zdefiniowanej funkcji* `implicit`, gdy wywołuje `son.implicit()` w linii 13., to nadal działa i wywołuje funkcję zdefiniowaną w klasie `Parent`. To pokazuje, że jeśli umieścisz funkcję w klasie bazowej (czyli `Parent`), wtedy wszystkie podklasy (na przykład `Child`) automatycznie pobiorą te funkcjonalności. Jest to bardzo przydatne w przypadku powtarzających się kodów, których potrzebujesz w wielu klasach.

Bezpośrednie nadpisanie

Problem posiadania funkcji, które są wywoływane domyślnie, polega na tym, że czasami chcesz, aby dziecko zachowywało się inaczej. W tym przypadku chcesz nadpisać tę funkcję w klasie potomnej, w efekcie zastępując funkcjonalność. Aby to zrobić, po prostu zdefiniuj w klasie `Child` funkcję o tej samej nazwie. Oto przykład.

ex44b.py

```
1  class Parent(object):
2
3      def override(self):
4          print("RODZIC override()")
5
6  class Child(Parent):
7
8      def override(self):
9          print("DZIECKO override()")
10
11  dad = Parent()
12  son = Child()
13
14  dad.override()
15  son.override()
```

W tym przykładzie mam funkcję o nazwie `override` w obu klasach, więc zobaczmy, co się stanie, kiedy ją uruchomię.

Ćwiczenie 44b — sesja

```
$ python3.6 ex44b.py
RODZIC override()
DZIECKO override()
```

Jak widać, wykonanie kodu w linii 14. powoduje uruchomienie funkcji `Parent.override`, ponieważ ta zmienna (`dad`) to `Parent`. Kiedy jednak zostaje uruchomiony kod z linii 15., drukowany jest komunikat funkcji `Child.override`, ponieważ `son` jest instancją klasy `Child`, a `Child` nadpisuje tę funkcję poprzez zdefiniowanie jej własnej wersji.

Zrób sobie teraz przerwę i zanim przejdiesz dalej, spróbuj pobawić się tymi dwiema koncepcjami.

Zmiana zachowania przed lub po

Trzecim sposobem korzystania z dziedziczenia jest szczególny przypadek nadpisania, w którym chcemy zmienić zachowanie funkcji przed uruchomieniem jej wersji z klasy `Parent` lub po. Najpierw nadpisujemy tę funkcję, tak jak w poprzednim przykładzie, ale potem używamy wbudowanej funkcji Pythona o nazwie `super`, aby pobrać do wywołania wersję z klasy `Parent`. Poniżej pokazany został przykład, który pomoże Ci zrozumieć ten opis.

ex44c.py

```
1  class Parent(object):
2
3      def altered(self):
4          print("RODZIC altered()")
5
6  class Child(Parent):
7
8      def altered(self):
9          print("DZIECKO PRZED altered() RODZICA")
10         super(Child, self).altered()
11         print("DZIECKO PO altered() RODZICA")
12
13  dad = Parent()
14  son = Child()
15
16  dad.altered()
17  son.altered()
```

Ważne są tutaj linie od 9. do 11., gdzie podczas wywołania `son.altered()` robię w klasie `Child` następujące rzeczy.

1. Ponieważ nadpisałem `Parent.altered`, uruchamiana jest wersja `Child.altered`, a linia 9. wykonywana jest tak, jakbyś tego oczekiwał.
2. W tym przypadku chcę zrobić zmianę przed i po, więc po linii 9. używam funkcji `super`, aby pobrać wersję `Parent.altered`.
3. W linii 10. wywołuję funkcję `super(Child, self).altered()`, która jest świadoma dziedziczenia i pobierze klasę `Parent`. Powinieneś odczytać to tak: „Wywołaj `super` z argumentami `Child` i `self`, a następnie wywołaj funkcję `altered` na tym, co zostanie zwrócone”.
4. W tym momencie uruchamiana jest wersja `Parent.altered` funkcji, która drukuje komunikat klasy `Parent`.
5. Wreszcie funkcja `Parent.altered` kończy wykonywanie i uruchamiana jest funkcja `Child.altered`, która drukuje komunikat „po”.

Jeśli uruchomisz ten kod, powinieneś zobaczyć następujące dane wyjściowe.

Ćwiczenie 44c — sesja

```
$ python3.6 ex44c.py
RODZIC altered()
DZIECKO PRZED altered() RODZICA
RODZIC altered()
DZIECKO PO altered() RODZICA
```

Połączenie wszystkich trzech sposobów

Aby zademonstrować wszystko jednocześnie, przygotowałem ostateczną wersję, która pokazuje każdy rodzaj interakcji dziedziczenia w jednym pliku.

ex44d.py

```
1  class Parent(object):
2
3      def override(self):
4          print("RODZIC override()")
5
6      def implicit(self):
7          print("RODZIC implicit()")
8
9      def altered(self):
10         print("RODZIC altered()")
11
12  class Child(Parent):
13
14      def override(self):
15         print("DZIECKO override()")
16
17      def altered(self):
18         print("DZIECKO PRZED altered() RODZICA")
19         super(Child, self).altered()
20         print("DZIECKO PO altered() RODZICA")
21
22  dad = Parent()
23  son = Child()
24
25  dad.implicit()
26  son.implicit()
27
28  dad.override()
29  son.override()
30
31  dad.altered()
32  son.altered()
```

Przeanalizuj każdą linię tego kodu i napisz komentarz wyjaśniający, co ona robi i czy jest nadpisaniem, czy nie. Następnie uruchom skrypt i potwierdź, czy otrzymujesz to, czego się spodziewałeś.

Ćwiczenie 44d — sesja

```
$ python3.6 ex44d.py
RODZIC implicit()
RODZIC implicit()
RODZIC override()
DZIECKO override()
RODZIC altered()
DZIECKO PRZED altered() RODZICA
RODZIC altered()
DZIECKO PO altered() RODZICA
```

Dlaczego super()

To wszystko powinno wydawać się logiczne i zgodne ze zdrowym rozsądkiem, ale wtedy wpadamy w kłopoty z czymś, co nazywa się dziedziczeniem wielokrotnym. Dziedziczenie wielokrotne występuje wtedy, gdy definiujesz klasę dziedziczącą po jednej lub po *wielu* klasach, na przykład:

```
class SuperFun(Child, BadStuff):  
    pass
```

To tak, jakby powiedzieć: „Utwórz klasę o nazwie SuperFun, która dziedziczy po klasach Child i BadStuff jednocześnie”.

W takim przypadku, gdy masz jakieś domyślne działania na dowolnej instancji SuperFun, Python musi sprawdzić możliwe funkcje w hierarchii klas zarówno dla Child, jak i BadStuff, ale musi to zrobić w konsekwentnej kolejności. W tym celu Python używa „kolejności rozwiązywania metod” (ang. *method resolution order* — MRO) i algorytmu o nazwie C3.

Ponieważ kolejność rozwiązywania metod jest złożona i używany jest ściśle określony algorytm, Python nie może pozostawić tych decyzji Tobie. Zamiast tego Python daje Ci funkcję `super()`, która zajmuje się tym wszystkim w miejscach, w których wymagane są działania zmieniające zachowanie, tak jak zrobiłem w przypadku `Child.altered`. Dzięki funkcji `super()` nie musisz się przejmować, czy zrobisz to prawidłowo, a Python znajdzie dla Ciebie odpowiednią funkcję.

Używanie super() z __init__

Najczęstszym zastosowaniem funkcji `super()` są funkcje `__init__` w klasach bazowych. Jest to zazwyczaj jedyne miejsce, w którym trzeba coś zrobić w klasie potomnej, a następnie zakończyć inicjowanie w klasie nadrzędnej. Oto krótki przykład, jak można to zrobić w klasie `Child`.

```
class Child(Parent):  
  
    def __init__(self, stuff):  
        self.stuff = stuff  
        super(Child, self).__init__()
```

Jest to prawie to samo, co poprzedni przykład `Child.altered`, z wyjątkiem ustawienia niektórych zmiennych w `__init__` przed zainicjowaniem `Parent` z `Parent.__init__`.

Kompozycja

Dziedziczenie jest przydatne, ale innym sposobem zrobienia tego samego jest użycie innych klas i modułów zamiast polegania na domyślnym dziedziczeniu. Jeśli przyjrzyś się trzem sposobom wykorzystywania dziedziczenia, zobaczysz, że dwa z nich obejmują pisanie nowego kodu w celu zastąpienia lub zmienienia funkcjonalności. Można to łatwo zreplikować, wywołując po prostu funkcje w module. Oto przykład.

ex44e.py

```
1  class Other(object):
2
3      def override(self):
4          print("override() klasy OTHER")
5
6      def implicit(self):
7          print("implicit() klasy OTHER")
8
9      def altered(self):
10         print("altered() klasy OTHER")
11
12  class Child(object):
13
14      def __init__(self):
15         self.other = Other()
16
17      def implicit(self):
18         self.other.implicit()
19
20      def override(self):
21         print("DZIECKO override()")
22
23      def altered(self):
24         print("DZIECKO PRZED altered() klasy OTHER")
25         self.other.altered()
26         print("DZIECKO PO altered() klasy OTHER")
27
28  son = Child()
29
30  son.implicit()
31  son.override()
32  son.altered()
```

W tym kodzie nie używam nazwy Parent, ponieważ *nie ma relacji* rodzic-dziecko. To jest relacja „ma”, gdzie Child ma Other, której używa, aby wykonać swoją pracę. Kiedy to uruchomię, otrzymam następujące dane wyjściowe.

Ćwiczenie 44e — sesja

```
$ python3.6 ex44e.py
implicit() klasy OTHER
DZIECKO override()
DZIECKO PRZED altered() klasy OTHER
altered() klasy OTHER
DZIECKO PO altered() klasy OTHER
```

Możesz zobaczyć, że większość kodu w klasach Child i Other jest taka sama, aby osiągnąć to samo. Jedyną różnicą jest to, że musiałem zdefiniować funkcję Child.implicit, aby wykonać to jedno działanie. Mógłbym wtedy zadać sobie pytanie, czy potrzebuję, aby Other było klasą i czy nie mógłbym po prostu umieścić tego w module o nazwie *other.py*?

Kiedy używać dziedziczenia, a kiedy kompozycji

Kwestia „dziedziczenie czy kompozycja” sprowadza się do próby rozwiązania problemu kodu wielokrotnego użytku. Na pewno nie chcesz mieć powielonego kodu w całym oprogramowaniu, ponieważ nie jest to czyste i wydajne. Dziedziczenie rozwiązuje ten problem, tworząc mechanizm umożliwiający korzystanie z domyślnych funkcji klas bazowych. Natomiast kompozycja rozwiązuje ten problem, dając Ci moduły i możliwość wywoływania funkcji z innych klas.

Jeśli obie metody rozwiązują problem ponownego użycia kodu, to które podejście jest odpowiednie w określonych sytuacjach? Odpowiedź jest niesamowicie subiektywna, ale dam Ci trzy wskazówki.

1. Unikaj wielokrotnego dziedziczenia za wszelką cenę, ponieważ jest zbyt skomplikowane, aby było niezawodne. Jeśli jednak nie będziesz miał wyjścia, przygotuj się na studiowanie hierarchii klas i szukanie, skąd pochodzą różne rzeczy.
2. Używaj kompozycji, aby spakować kod w moduły wykorzystywane w wielu różnych, niepowiązanych ze sobą miejscach i sytuacjach.
3. Używaj dziedziczenia tylko wtedy, gdy istnieją wyraźnie powiązane fragmenty kodu wielokrotnego użytku, które pasują do pojedynczej wspólnej koncepcji, lub jeśli musisz to zrobić z powodu czegoś, z czego korzystasz.

Nie bądź niewolnikiem tych zasad. Należy pamiętać, że programowanie obiektowe jest całkowicie społeczną konwencją opracowaną przez programistów do pakowania i udostępniania kodu. Ponieważ jest to konwencja społeczna, ale skodyfikowana w Pythonie, możesz być zmuszony do łamania tych zasad z uwagi na ludzi, z którymi pracujesz. W takim przypadku dowiedz się, w jaki sposób korzystają z różnych rzeczy, a następnie dostosuj się do sytuacji.

Zrób to sam

W tym ćwiczeniu jest tylko jeden punkt „Zrób to sam”, ponieważ to dość obszerne ćwiczenie. Przeczytaj dokumentację ze strony <http://www.python.org/dev/peps/pep-0008/> i zacznij używać tych wskazówek w swoim kodzie. Zauważysz, że niektóre z nich różnią się tego, czego uczysz się w tej książce, ale teraz powinieneś być w stanie zrozumieć te rekomendacje i zacząć używać ich we własnym kodzie. Reszta kodu w tej książce może, ale nie musi, być zgodna z tymi wytycznymi, w zależności od tego, czy nie sprawia, iż kod staje się bardziej zagmatwany. Zalecam Ci to również dlatego, że zrozumienie jest ważniejsze od imponowania wszystkim dookoła wiedzą o ezoterycznych regułach stylu.

Typowe pytania

Jak mogę zwiększyć swoją sprawność w rozwiązywaniu problemów, z którymi wcześniej się nie spotkałem? Jedynym sposobem na lepsze radzenie sobie z rozwiązywaniem problemów jest *samodzielne* rozpracowywanie jak największej liczby problemów. Zazwyczaj, gdy ludzie natrafiają na trudny problem, od razu zaczynają szukać podpowiedzi. Nie ma w tym niczego złego, kiedy musisz wykonać

określone zadanie, ale jeśli masz czas na rozwiązanie danego problemu samodzielnie, poświęć ten czas. Zatrzymaj się i mierz z problemem tak długo, jak to możliwe, próbując wszelkich możliwych rzeczy, dopóki go nie rozwiążesz lub nie poddasz się. Po tym odpowiedzi, które znajdziesz, będą bardziej satysfakcjonujące, a w rezultacie poprawisz swoje umiejętności rozwiązywania problemów.

Czy obiekty nie są po prostu kopiami klas? W niektórych językach (takich jak JavaScript) jest to prawda. Takie języki są zwane językami prototypowymi i poza użyciem nie ma w nich większej różnicy między obiektami i klasami. Jednak w Pythonie klasy działają jak szablony, które „wybijają” nowe obiekty, podobnie jak bije się monety za pomocą matrycy (szablону).

Tworzysz grę

Musisz zacząć uczyć się sam. Mam nadzieję, że podczas pracy z tą książką przekonałeś się, iż wszystkie potrzebne informacje znajdują się w internecie. Musisz po prostu ich poszukać. Brakowało Ci tylko właściwych słów i świadomości tego, czego należy szukać. Teraz powinieneś już to rozumieć, więc najwyższy czas, abyś przebrnął przez wielki projekt i spróbował go uruchomić.

Oto Twoje wymagania.

1. Opracuj grę inną od tej, którą napisałem.
2. Użyj więcej niż jednego pliku i posłuż się instrukcją `import`, aby z nich korzystać. Upewnij się, że wiesz, co to jest.
3. Użyj *jednej klasy na pokój* i nazwij te klasy odpowiednio do ich przeznaczenia (na przykład pokój ze złotem możesz nazwać `ZłotyPokoj`).
4. Twój gracz będzie musiał wiedzieć o istnieniu tych pokoi, więc utwórz klasę, która je uruchamia i wie o nich. Można to zrobić na wiele sposobów, ale zastanów się nad tym, aby każdy pokój zwracał pokój, który jest następny, lub ustawiał zmienną wskazującą kolejny pokój.

O pozostałych rzeczach zdecyduj sam. Poświęć na to cały tydzień i postaraj się, żeby była to najlepsza gra, jaką potrafisz napisać. Używaj klas, funkcji, słowników, list i wszystkiego, czego możesz, żeby było fajnie. Celem tej lekcji jest to, byś nauczył się tworzyć struktury klas, które potrzebują innych klas znajdujących się w innych plikach.

Pamiętaj, że nie mówię *dokładnie*, jak to zrobić, ponieważ musisz to wykonać sam. Wymyśl to. Programowanie to rozwiązywanie problemów, a to oznacza próbowanie różnych rzeczy, eksperymentowanie, niepowodzenia, wyrzucanie całej pracy do kosza i zaczynanie od nowa. Kiedy utkniesz, poproś o pomoc i pokaż komuś swój kod. Jeśli dana osoba będzie dla Ciebie niemiła, zignoruj ją i skupiaj się na ludziach, którzy nie zachowują się w ten sposób i oferują pomoc. Kontynuuj pracę i czyść rozwiązanie, aż będzie wystarczająco dobre, żeby je znowu komuś pokazać.

Powodzenia i do zobaczenia za tydzień z gotową grą.

Ocenianie napisanej gry

W tym ćwiczeniu ocenisz grę, którą właśnie przygotowałeś. Może doszedłeś do jakiegoś etapu i utknąłeś. Może gra działa, ale ledwo, ledwo. Tak czy inaczej przyjrzymy się kilku rzeczom, o których powinieneś już wiedzieć, i upewnimy się, że uwzględniłeś je w swojej grze. Będziemy się uczyć poprawnie formatować klasy, stosować konwencje typowe podczas używania klas i przyswoimy dużo „podręcznikowej” wiedzy.

Dlaczego kazałem Ci spróbować zrobić to samodzielnie, a następnie pokazuję, jak to wykonać? Od tej chwili będziesz próbował wybić się na samodzielność. Będę Cię do tego zmuszał. Przez cały ten czas przeważnie trzymałem Cię za rękę i dłużej tego nie będę robić.

Teraz będę zadawał zadania do zrobienia, Ty będziesz wykonywał je samodzielnie, a potem pokażę Ci sposoby poprawienia tego, co zrobiłeś.

Z początku będziesz pracował w pocie czoła i prawdopodobnie będziesz bardzo sfrustrowany, ale trzymaj się, a w końcu przyzwyczaisz umysł do rozwiązywania problemów. Zacznieś znajdować twórcze rozwiązania problemów, a nie tylko kopiować rozwiązania z podręczników.

Styl funkcji

Do tej kategorii zaliczają się wszystkie reguły dotyczące tworzenia przyjaznych funkcji, których nauczyłem Cię do tej pory, oraz kilka nowych.

- Z różnych powodów programiści nazywają funkcje należące do klas „metodami”. To głównie marketing, ale pamiętaj, że za każdym razem, gdy powiesz „funkcja”, będą denerwować Cię poprawiać i mówić, że jest to „metoda”. Jeśli zaczną być zbyt irytujący, poproś ich, aby zademonstrowali podstawy matematyczne, które określają, w jaki sposób „metoda” różni się od „funkcji”, a wtedy zamilkną.
- Gdy pracujesz z klasami, dużo czasu poświęcasz na rozmawianie o tym, jak sprawić, żeby klasa „robiła jakieś rzeczy”. Zamiast nazywać funkcję na podstawie tego, co ona robi, nazwij ją tak, jakby było to polecenie, które dajesz klasie. Przykładowo pop mówi: „Hej, listo, usuń ten element z końca”. Nie nazywa się to `remove_from_end_of_list`, ponieważ nie jest to **polecenie** listy, mimo że to właśnie robi.
- Zachowuj niewielki rozmiar i prostotę funkcji. Z jakiegoś powodu, kiedy ludzie zaczynają się uczyć o klasach, zapominają o tym.

Styl klas

- Twoje klasy powinny używać notacji typu *camelCase*, tak jak w nazwie `SuperGoldFactory`, zamiast „formatu z podkreślnikami”, tak jak w `super_gold_factory`.
- Staraj się nie robić zbyt wiele w funkcjach `__init__`. To sprawia, że są trudniejsze w użyciu.
- Twoje pozostałe funkcje powinny używać formatu z podkreślnikami, więc pisz `my_awesome_hair`, a nie `myawesomehair` lub `MyAwesomeHair`.
- Zachowaj spójność w sposobie porządkowania argumentów funkcji. Jeśli Twoja klasa ma do czynienia z użytkownikami (`user`), psami (`dog`) i kotami (`cat`), zachowuj ten porządek przez cały czas, chyba że naprawdę nie będzie miało to sensu. Jeżeli posiadasz funkcję, która przyjmuje (`dog`, `cat`, `user`), a druga przyjmuje (`user`, `cat`, `dog`), będzie to trudne w użyciu.
- Staraj się nie używać zmiennych pochodzących z modułu lub globalnych. Zmienne powinny być możliwie niezależne.
- Bezrefleksyjnie stosowana spójność to chochlik małych umysłów. Spójność jest dobra, ale głupie podążanie za jakąś idiotyczną mantrą, bo wszyscy inni tak robią, to zły styl. Myśl samodzielnie.

- Zawsze, ale to zawsze używaj formatu `class Nazwa(object)`, bo inaczej będziesz miał poważne kłopoty.

Styl kodu

- Stosuj w kodzie pionowe odstępę, aby łatwiej było go czytać. Spotkasz się z pewnymi bardzo złymi programistami, którzy są w stanie napisać rozsądny kod, ale nie dodają *żadnych* odstępów. Jest to zły styl w każdym języku, ponieważ ludzkie oko i mózg wykorzystują odstępę i pionowe wyrównanie do skanowania i oddzielania wizualnych elementów. Brak odstępów jest tym samym, co „pomalowanie” kodu w barwy doskonałego kamuflażu.
- Jeśli nie możesz przeczytać kodu na głos, prawdopodobnie trudno go odczytać. Jeśli miewasz problemy z tworzeniem czegoś łatwego w użyciu, spróbuj odczytać to na głos. Zmusi Cię to nie tylko do zwolnienia tempa i rzeczywistego przeczytania kodu, ale pomoże również znaleźć trudne fragmenty i rzeczy, które należy zmienić dla lepszej czytelności.
- Dopóki nie znajdziesz własnego stylu, próbuj robić w Pythonie to, co robią inni.
- Gdy odnajdziesz swój własny styl, postaraj się z nim nie przesadzać. Praca z kodami innych ludzi jest częścią bycia programistą, a inni ludzie miewają naprawdę zły gust. Zaufaj mi, prawdopodobnie również masz zły smak i nawet nie zdajesz sobie z tego sprawy.
- Jeśli znajdziesz kogoś, kto pisze kod w stylu, który Ci się podoba, spróbuj napisać coś, co naśladuje ten styl.

Dobre komentarze

- Programiści powiedzą, że Twój kod powinien być na tyle czytelny, żebyś nie potrzebował komentarzy. Następnie powiedzą swoim najbardziej oficjalnym głosem: „Ergo, nigdy nie powinno się pisać komentarzy lub dokumentacji. CO BYŁO DO OKAZANIA”. Ci programiści są albo konsultantami, którzy dostają więcej pieniędzy, jeśli inne osoby nie są w stanie używać ich kodu, lub są po prostu niekompetentni i nigdy nie pracują z innymi ludźmi. Ignoruj ich i pisz komentarze.
- Kiedy piszesz komentarze, opisuj, *dłaczego* robisz to, co robisz. Kod mówi już „w jaki sposób”, ważniejsze jest więc to, dlaczego zrobiłeś coś tak, a nie inaczej.
- Kiedy piszesz komentarze dokumentujące do Twoich funkcji, przygotuj dokumentację pod kątem kogoś, kto będzie musiał użyć Twojego kodu. Nie musisz szaleć, ale jedno miłe zdanie o tym, co ktoś może zrobić z tą funkcją, bardzo pomaga.
- Chociaż komentarze są dobre, zbyt duża ich liczba jest czymś złym, a ponadto trzeba je utrzymywać. Niech komentarze będą stosunkowo krótkie i na temat, a jeśli zmienisz funkcję, przejrzyj komentarz, aby upewnić się, że wciąż jest poprawny.

Oceń swoją grę

Chcę, żebyś teraz udawał, że jesteś mną. Przyjmij bardzo surowy wygląd, wydrukuj swój kod, weź czerwony długopis i oznacz każdy błąd, który znajdziesz, włącznie z różnymi kwestiami z tego ćwiczenia i innymi wskazówkami, jakie poznałeś do tej pory. Gdy skończysz oznaczanie kodu, chcę, abyś naprawił wszystko, co znalazłeś. Następnie powtórz to kilka razy, szukając czegoś, co można jeszcze poprawić. Wykorzystaj wszystkie sztuczki, które Ci pokazałem, aby poddać kod możliwie najbardziej drobiazgowej, najdrobniejszej analizie, jaką możesz wykonać.

Celem tego ćwiczenia jest trening skupiania uwagi na szczegółach dotyczących klas. Gdy skończysz z tym fragmentem kodu, znajdź kod jakiegoś innego programisty i zrób to samo. Przeanalizuj drukowaną kopię jakiejś jego części, wskaż wszystkie pomyłki i błędy stylu. Następnie napraw kod i sprawdź, czy te poprawki można wprowadzić bez popsucia programu.

Chcę, żebyś przez cały tydzień wyłącznie oceniał i naprawiał kod — własny i cudzy. Będzie to dość ciężka praca, ale kiedy skończysz, Twój mózg będzie zwarty i gotowy jak pięści boksera.

Szkielet projektu

Teraz zaczniesz się uczyć, jak skonfigurować dobry katalog „szkieletu” projektu. Ten szkieletowy katalog będzie zawierał wszystkie podstawowe rzeczy potrzebne do przygotowania i uruchomienia nowego projektu. Będzie miał Twój układ projektu, zautomatyzowane testy, moduły i skrypty instalacyjne. Kiedy będziesz zabierał się do jakiegoś nowego projektu, na początek po prostu skopiuj ten katalog do nowej lokalizacji z nową nazwą i wyedytuj pliki.

Konfiguracja w systemach macOS i Linux

Zanim zaczniesz to ćwiczenie, musisz zainstalować oprogramowanie dla Pythona. Przy użyciu narzędzia o nazwie `pip3.6` (lub po prostu `pip`) zainstalujesz nowe moduły. Polecenie `pip3.6` powinno być dołączone do Twojej instalacji Pythona 3.6. Możesz to zweryfikować za pomocą tego polecenia:

```
$ pip3.6 list
pip (9.0.1)
setuptools (28.8.0)
$
```

Możesz zignorować wszelkie ostrzeżenia o przestarzałym formacie, jeśli je zobaczysz. Możesz także zobaczyć zainstalowane inne narzędzia, ale podstawowe powinny być `pip` i `setuptools`. Gdy już to zweryfikujesz, możesz zainstalować `virtualenv`:

```
$ sudo pip3.6 install virtualenv
Password:
Collecting virtualenv
  Downloading virtualenv-15.1.0-py2.py3-none-any.whl (1.8MB)
    100% ||||| 1.8MB 1.1MB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-15.1.0
$
```

Dotyczy to systemów Linux lub macOS. Jeśli korzystasz z jednego z tych systemów, uruchom poniższe polecenie, aby upewnić się, że używasz poprawnego `virtualenv`:

```
$ whereis virtualenv
/Library/Frameworks/Python.framework/Versions/3.6/bin/virtualenv
```

W systemie macOS powinieneś zobaczyć listing podobny do powyższego, ale w Linuksie będzie inaczej. W systemie Linux możesz mieć rzeczywiste polecenie `virtualenv3.6` lub jeszcze lepiej będzie, jak zainstalujesz pakiet dla niego z systemu zarządzania pakietami.

Po zainstalowaniu `virtualenv` możesz go użyć do utworzenia wirtualnego środowiska Pythona, które ułatwia zarządzanie wersjami pakietów dla różnych projektów. Najpierw uruchom poniższe polecenia, a za chwilę wyjaśnię, co one robią.

```
$ mkdir ~/.venvs
$ virtualenv --system-site-packages ~/.venvs/lpthw
$ . ~/.venvs/lpthw/bin/activate
(lpthw) $
```

Oto, co się tutaj dzieje linia po linii.

1. Tworzysz katalog o nazwie `.venvs` w swoim *HOME* `~/`, aby przechowywać wszystkie wirtualne środowiska.
2. Uruchamiasz `virtualenv` i instruujesz, aby załączył pakiety dostępne w systemie dla podstawowej instalacji Pythona (`--system-site-packages`), a następnie instruujesz, aby zbudował środowisko wirtualne w `~/.venvs/lpthw`.
3. Następnie wskazujesz ścieżkę do wirtualnego środowiska `lpthw` przy użyciu operatora `.` powłoki bash, po którym podajesz skrypt `~/.venvs/lpthw/bin/activate`.
4. Na koniec zmienia się nagłówek wiersza poleceń na `(lpthw)`, więc wiesz, że używasz właśnie tego wirtualnego środowiska.

Teraz możesz zobaczyć, gdzie zostały zainstalowane te rzeczy.

```
(lpthw) $ which python
/Users/zedshaw/.venvs/lpthw/bin/python
(lpthw) $ python
Python 3.6.0rc2 (v3.6.0rc2:800a67f7806d, Dec 16 2016, 14:12:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
(lpthw) $
```

Możesz zauważyć, że uruchamiany python jest zainstalowany w katalogu `/Users/zedshaw/.venvs/lpthw/bin/python` zamiast w oryginalnej lokalizacji. To rozwiązuje również problem konieczności wpisywania polecenia `python3.6`, ponieważ instalują się oba.

```
$ which python3.6
/Users/zedshaw/.venvs/lpthw/bin/python3.6
(lpthw) $
```

To samo dotyczy poleceń `virtualenv` i `pip`. Ostatnim etapem tej konfiguracji jest instalacja `nose`, frameworku testowego, którego użyjemy w tym ćwiczeniu.

```
$ pip install nose
Collecting nose
  Downloading nose-1.3.7-py3-none-any.whl (154kB)
    100% |#####| 163kB 3.2MB/s
Installing collected packages: nose
Successfully installed nose-1.3.7
(lpthw) $
```

Konfiguracja w systemie Windows 10

Instalacja w systemie Windows 10 jest nieco prostsza niż w systemach Linux lub macOS, ale tylko wtedy, gdy masz zainstalowaną *jedną* wersję Pythona. Jeśli masz zainstalowane wersje zarówno Python 3.6, jak i Python 2.7, jesteś zdany na siebie, ponieważ zarządzanie wieloma

instalacjami jest trudne. Jeśli do tej pory wykonywałeś prawidłowo polecenia z tej książki i masz tylko Pythona 3.6, zrobisz to, co teraz opiszę. Najpierw przejdź do swojego katalogu głównego i sprawdź, czy używasz właściwej wersji Pythona.

```
> cd ~
> python
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12)
  [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
```

Następnie uruchom pip, aby potwierdzić, że masz podstawową instalację.

```
> pip list
pip (9.0.1)
setuptools (28.8.0)
```

Możesz bezpiecznie zignorować wszelkie ostrzeżenia o przestarzałym formacie i nie ma problemu, jeśli masz zainstalowane inne pakiety. Potem instalujesz virtualenv do konfiguracji prostych wirtualnych środowisk, bo będzie potrzebne dalej w tej książce.

```
> pip install virtualenv
Collecting virtualenv
  Using cached virtualenv-15.1.0-py2.py3-none-any.whl
Installing collected packages: virtualenv
Successfully installed virtualenv-15.1.0
```

Po zainstalowaniu virtualenv będziesz musiał utworzyć katalog `.venvs` i wypełnić go wirtualnym środowiskiem.

```
> mkdir .venvs
> virtualenv --system-site-packages .venvs/lpthw
Using base prefix
  'c:\users\zedshaw\appdata\local\programs\python\python36'
New python executable in
  C:\Users\zedshaw\.venvs\lpthw\Scripts\python.exe
Installing setuptools, pip, wheel...done.
```

Te dwa polecenia tworzą folder `.venvs` do przechowywania różnych środowisk wirtualnych, a następnie Twoje pierwsze środowisko wirtualne o nazwie `lpthw`. Środowisko wirtualne (virtualenv) to „fałszywe” miejsce do instalowania oprogramowania, żebyś mógł mieć oddzielne wersje różnych pakietów dla każdego projektu, nad którym pracujesz. Kiedy masz skonfigurowane środowisko wirtualne, musisz je aktywować.

```
> .\.venvs\lpthw\Scripts\activate
```

To polecenie uruchomi skrypt aktywacyjny dla PowerShell, który konfiguruje środowisko wirtualne `lpthw` dla bieżącej powłoki. Za każdym razem, gdy chcesz korzystać z oprogramowania do książki, uruchamiasz to polecenie. Zauważysz przy następnym poleceniu, że teraz do nagłówka wiersza poleceń PowerShell dodane jest (`lpthw`), co pokazuje, którego virtualenv używasz. Na koniec wystarczy zainstalować nose w celu późniejszego uruchamiania testów.

```
(lpthw) > pip install nose
Collecting nose
```



```

Downloading nose-1.3.7-py3-none-any.whl (154kB)
100% |#####| 163kB 1.2MB/s
Installing collected packages: nose
Successfully installed nose-1.3.7
(lpthw) >

```

Zobaczysz, że to instaluje nose, z tym że pip zainstaluje ten framework w Twoim wirtualnym środowisku `.venvs\lpthw` zamiast w głównym katalogu pakietów systemowych. Pozwala to na instalowanie konfliktowych wersji pakietów Pythona dla każdego projektu, na którym pracujesz, bez infekowania głównej konfiguracji systemu.

Tworzenie szkieletu katalogu projektów

Najpierw utwórz strukturę Twojego katalogu szkieletowego za pomocą poniższych poleceń.

```

$ mkdir projects
$ cd projects/
$ mkdir skeleton
$ cd skeleton
$ mkdir bin NAME tests docs

```

Używam katalogu o nazwie *projects* do przechowywania wszystkich rzeczy, nad którymi pracuję. Wewnątrz tego katalogu mam katalog *skeleton*, w którym umieszczam podstawę moich projektów. Nazwa katalogu *NAME* zostanie zmieniona na taką, której używasz dla głównego modułu projektu, kiedy korzystasz z tego szkieletu.

Następnie musimy skonfigurować kilka początkowych plików. Tak należy to zrobić w systemach Linux i macOS.

```

$ touch NAME/__init__.py
$ touch tests/__init__.py

```

To samo dotyczy programu PowerShell w systemie Windows.

```

$ new-item -type file NAME/__init__.py
$ new-item -type file tests/__init__.py

```

Tworzymy pusty katalog modułów Pythona, w którym możemy umieścić nasz kod. Następnie musimy utworzyć plik *setup.py*, którego później użyjemy, aby zainstalować nasz projekt, jeśli będzie trzeba.

`setup.py`

```

1  try:
2      from setuptools import setup
3  except ImportError:
4      from distutils.core import setup
5
6  config = {
7      'description': 'Mój projekt',
8      'author': 'Moje nazwisko',
9      'url': 'Adres URL, z którego można go pobrać.',

```

```

10     'download_url': 'Gdzie go można pobrać.',
11     'author_email': 'Mój email.',
12     'version': '0.1',
13     'install_requires': ['nose'],
14     'packages': ['NAME'],
15     'scripts': [],
16     'name': 'nazwa projektu'
17 }
18
19 setup(**config)

```

Wyedytuj ten plik, aby zawierał Twoje dane kontaktowe; po skopiowaniu będzie gotowy do pracy.

Na koniec będziesz potrzebował dla testów prostego pliku szkieletu o nazwie *tests/NAME_tests.py*.

NAME_tests.py

```

1  from nose.tools import *
2  import NAME
3
4  def setup():
5      print("KONFIGURACJA!")
6
7  def teardown():
8      print("ZAMYKANIE!")
9
10 def test_basic():
11     print("URUCHOMIONO TEST!")

```

Ostateczna struktura katalogów

Kiedy skończysz konfigurowanie, Twój katalog powinien wyglądać tak, jak mój tutaj.

```

skeleton/
  NAME/
    __init__.py
  bin/
  docs/
  setup.py
  tests/
    NAME_tests.py
    __init__.py

```

Od teraz powinieneś uruchamiać swoje polecenia z tego katalogu. Jeśli nie możesz, wykonaj polecenie `ls -R` i jeśli nie widzisz tej samej struktury, to znaczy, że jesteś w niewłaściwym miejscu. Ludzie zwykle wchodzi na przykład do katalogu *tests/*, aby spróbować uruchomić tam pliki, co nie będzie działać. Aby uruchamiać testy aplikacji, musisz być *nad* folderem *tests/* i tę lokalizację pokazałem powyżej. Tak więc, jeśli spróbujesz zrobić w ten sposób:

```
$ cd tests/      # ŻŁE! ŻŁE! ŻŁE!
$ nosetests
```

```
-----
Ran 0 tests in 0.000s
```

```
OK
```

...to będzie źle! Musisz być nad folderem `tests/`, więc zakładając, że popełniłeś ten błąd, naprawisz to, kiedy zrobisz tak:

```
$ cd ..      # wyjdź z folderu tests/
$ ls         # PRAWDŁOWO! jesteś teraz we właściwym miejscu
NAME        bin          docs          setup.py      tests
$ nosetests
.
```

```
-----
Ran 1 test in 0.004s
```

```
OK
```

Pamiętaj o tym, ponieważ ludzie często popełniają ten błąd.

OSTRZEŻENIE! W czasie publikacji dowiedziałem się, że projekt nose został zarzucony i może nie działać prawidłowo. Jeśli otrzymujesz dziwne błędy składniowe podczas uruchamiania `nosetests`, przyjrzyj się informacjom o błędzie. Jeżeli odwołują się one do „python2.7”, istnieje prawdopodobieństwo, że `nosetests` próbuje uruchomić na Twoim komputerze wersję 2.7 Pythona. Rozwiązaniem w systemach macOS lub Linux jest uruchomienie frameworku nose za pomocą polecenia `python3.6 -m "nose"`. W systemie Windows możesz nie mieć tego problemu, ale jeśli się pojawi, rozwiąże go użycie polecenia `python -m "nose"`.

Testowanie konfiguracji

Po zainstalowaniu wszystkiego powinieneś być w stanie zrobić tak:

```
$ nosetests
```

```
.
```

```
-----
Ran 1 test in 0.007s
```

```
OK
```

Wyjaśnię, co robi polecenie `nosetests` w następnym ćwiczeniu, ale na razie, jeśli nie widzisz powyższego listingu, prawdopodobnie zrobiłeś coś źle. Upewnij się, że umieściłeś pliki `__init__.py` w folderach `NAME` i `tests` oraz że plik `tests/NAME_tests.py` jest prawidłowy.

Używanie szkieletu

Masz już za sobą większość katorzniczej pracy. Ilekróć będziesz chciał rozpocząć nowy projekt, po prostu wykonaj następujące czynności.

1. Zrób kopię swojego katalogu szkieletowego. Nazwij go tak, jak ma nazywać się Twój nowy projekt.
2. Zmień nazwę (przenieś) katalogu *NAME* na nazwę projektu lub na taką, jakiej chcesz używać dla modułu głównego.
3. Wyedytuj plik *setup.py*, aby posiadał wszystkie informacje dotyczące projektu.
4. Zmień nazwę pliku *tests/NAME_tests.py*, aby również miał nazwę modułu.
5. Sprawdź ponownie, czy wszystko działa, używając jeszcze raz polecenia *nosetests*.
6. Rozpocznij kodowanie.

Wymagany quiz

To ćwiczenie nie ma podrozdziału „Zrób to sam”. Zamiast tego jest quiz, który powinienś rozwiązać.

1. Przeczytaj, jak korzystać ze wszystkich rzeczy, które zainstalowałeś.
2. Poczytaj o pliku *setup.py* i wszystkim, co ma do zaoferowania. Ostrzeżenie: nie jest to najlepiej napisany fragment oprogramowania, więc będzie bardzo dziwny w użyciu.
3. Utwórz projekt i zacznij umieszczać kod w module, a następnie uruchom moduł.
4. Umieść w katalogu *bin* skrypt, który możesz uruchomić. Przeczytaj, jak możesz utworzyć skrypt Pythona, który jest uruchamialny dla Twojego systemu.
5. W pliku *setup.py* wpisz nazwę utworzonego skryptu *bin*, aby został zainstalowany.
6. Użyj swojego pliku *setup.py*, aby zainstalować własny moduł, i upewnij się, że działa, a następnie użyj polecenia *pip*, żeby usunąć jego instalację.

Typowe pytania

Czy te instrukcje działają w systemie Windows? Powinny, ale w zależności od wersji systemu Windows być może będziesz musiał trochę powalczyć z konfiguracją, aby to zadziałało. Po prostu szukaj informacji i próbuj, aż Ci się uda, albo jeśli masz taką możliwość, poproś o pomoc znajomego bardziej doświadczonego w kwestiach Pythona i systemu Windows.

Co mam umieścić w słowniku *config* w moim pliku *setup.py*? Poczytaj dokumentację dla *distutils* na stronie <http://docs.python.org/distutils/setupscript.html>.

Nie mogę załadować modułu *NAME* i po prostu dostaję błąd `ImportError`. Upewnij się, że utworzyłeś plik *NAME/__init__.py*. Jeśli korzystasz z systemu Windows, upewnij się, że przypadkowo nie nazwałeś go *NAME__init__.py.txt*, co dzieje się domyślnie w niektórych edytorach.

Dlaczego w ogóle potrzebujemy folderu *bin*? Jest to standardowe miejsce do umieszczania skryptów uruchamianych w wierszu poleceń, a nie miejsce na moduły.

Gdy uruchamiam *nosetests*, pokazuje, że uruchamiany jest tylko jeden test. Czy to prawidłowe? Tak, moje dane wyjściowe pokazują to samo.

Zautomatyzowane testowanie

Ciągłe wpisywanie poleceń w grze, aby upewnić się, że gra działa, jest irytujące. Czy nie lepiej byłoby napisać niewielkich fragmentów kodu, które testują kod gry? Wtedy przy wprowadzaniu zmian lub dodawaniu nowych rzeczy do programu można by po prostu „uruchomić testy”, które zapewniałyby, że wszystko wciąż działa. Te zautomatyzowane testy nie wychwycą wszystkich błędów, ale skrócą czas poświęcony na kilkakrotne wpisywanie poleceń i uruchamianie kodu.

Wszystkie następne ćwiczenia zamiast podrozdziału „Co powinieneś zobaczyć” będą zawierały podrozdział „Co powinieneś przetestować”. Od teraz będziesz pisał zautomatyzowane testy dla całego kodu i mam nadzieję, że to sprawi, iż staniesz się jeszcze lepszym programistą.

Nie będę próbował wyjaśniać, dlaczego powinieneś pisać zautomatyzowane testy. Powiem tylko, że próbujesz być programistą, a programiści automatyzują nudne i żmudne zadania. Testowanie oprogramowania jest zdecydowanie nudne i nużące, więc równie dobrze możesz napisać trochę kodu, który zrobi to za Ciebie.

To wyjaśnienie powinno wystarczyć, ponieważ *Twoim* powodem pisania testów jednostkowych jest ćwiczenie umysłu. Powinieneś czytać tę książkę, pisząc jednocześnie kod, który robi różne rzeczy. Teraz pójdziesz krok dalej i napiszesz kod, który „wie” o innym napisanym przez Ciebie kodzie. Proces pisania testów uruchamiających napisany przez Ciebie kod *zmusza* do właściwego zrozumienia tego, co właśnie napisałeś. Utrwala w głowie, co robi napisany kod i dlaczego działa oraz zapewnia nowy poziom skupienia się na szczegółach.

Pisanie przypadku testowego

Weźmiemy bardzo prosty fragment kodu i napiszemy jeden prosty test. Oprzemy ten niewielki test na nowym projekcie ze szkieletu projektu.

Najpierw utwórz projekt `ex47` na podstawie szkieletu projektu. Poniżej wymienione zostały czynności, które powinieneś wykonać. Zamiast pokazywać, jak to robić w praktyce, opiszę po prostu instrukcje, abyś mógł dojść do tego samodzielnie.

1. Skopiuj zawartość folderu *skeleton* do folderu `ex47`.
2. Zmień nazwy wszystkich wystąpień *NAME* w nazwach folderów i plików na `ex47`.
3. We wszystkich plikach zmień słowo *NAME* na `ex47`.
4. Na koniec usuń wszystkie pliki **.pyc*, aby upewnić się, że masz czysty szkielet projektu.

Jeśli na czymś utkniesz, odwołaj się do ćwiczenia 46., a jeśli słabo będzie Ci szło, pocwicz kilka razy.

OSTRZEŻENIE! Pamiętaj, że w celu uruchomienia testów wpisujesz polecenie `nosetests`. Możesz też uruchomić je za pomocą polecenia `python3.6 ex47_tests.py`, ale nie będzie działało tak łatwo i będziesz musiał robić tak dla każdego pliku testowego.

Następnie utwórz prosty plik `ex47/ex47/game.py`, w którym umieścisz kod do przetestowania. To będzie głupiotka mała klasa z następującym kodem.

game.py

```
1  class Room(object):
2
3      def __init__(self, name, description):
4          self.name = name
5          self.description = description
6          self.paths = {}
7
8      def go(self, direction):
9          return self.paths.get(direction, None)
10
11     def add_paths(self, paths):
12         self.paths.update(paths)
```

Gdy już wpiszesz ten plik, zmień szkielet testu jednostkowego na następujący kod.

ex47_tests.py

```
1  from nose.tools import *
2  from ex47.game import Room
3
4
5  def test_room():
6      gold = Room("GoldRoom",
7                  """W tym pokoju jest złoto, które możesz zabrać.
8                     Na północy są drzwi.""")
9      assert_equal(gold.name, "GoldRoom")
10     assert_equal(gold.paths, {})
11
12     def test_room_paths():
13         center = Room("Center", "Test pokoju pośrodku.")
14         north = Room("North", "Test pokoju na północy.")
15         south = Room("South", "Test pokoju na południu.")
16
17         center.add_paths({'north': north, 'south': south})
18         assert_equal(center.go('north'), north)
19         assert_equal(center.go('south'), south)
20
21     def test_map():
22         start = Room("Start", "Możesz iść na zachód i w dół.")
23         west = Room("Trees", "Tutaj są trzy drzewa, możesz iść na wschód.")
24         down = Room("Dungeon", "Tu na dole jest ciemno, możesz iść do góry.")
25
```

```
26     start.add_paths({'west': west, 'down': down})
27     west.add_paths({'east': start})
28     down.add_paths({'up': start})
29
30     assert_equal(start.go('west'), west)
31     assert_equal(start.go('west').go('east'), start)
32     assert_equal(start.go('down').go('up'), start)
```

Ten plik importuje klasę `Room`, którą utworzyłeś w module `ex47.game`, żebyś mógł wykonywać na niej testy. Dalej masz zestaw testów, które są funkcjami zaczynającymi się od `test_`. Wewnątrz każdego przypadku testowego znajduje się nieco kodu, który tworzy pokój lub zestaw pokoi, a następnie sprawdza, czy pokoje działają zgodnie z oczekiwaniami. Testuje podstawowe funkcjonalności pokoju, następnie ścieżki, a potem wypróbowuje całą mapę.

Ważnymi funkcjami są tutaj funkcje `assert_equal`, które sprawdzają, czy ustawione zmienne lub ścieżki, które zbudowałeś w klasie `Room`, są w rzeczywistości tym, czym Ci się zdaje. Jeśli otrzymasz zły wynik, `nosetests` wydrukują komunikat o błędzie, abyś mógł zorientować się, co jest nie tak.

Wytyczne testowania

Podczas przeprowadzania testów postępuj zgodnie z poniższym zestawem wskazówek.

1. Pliki testowe są umieszczane w folderze `tests/` i nazywają się `CÓSTAM_tests.py`, w przeciwnym razie `nosetests` ich nie uruchomi. Ponadto dzięki temu testy nie będą kolidować z innym kodem.
2. Napisz jeden plik testowy dla każdego tworzonego modułu.
3. Niech Twoje przypadki testowe (funkcje) będą krótkie, ale nie przejmuj się, jeśli są trochę niechlujne. Przypadki testowe są zwykle trochę niechlujne.
4. Chociaż przypadki testowe są niechlujne, staraj się utrzymywać je w czystości i usuwaj w miarę możliwości każdy powtarzający się kod. Utwórz funkcje pomocnicze, które pozwolą pozbyć się zduplikowanego kodu. Podziękujesz mi później, kiedy po jakiejś zmianie będziesz musiał zmienić testy. Zduplikowany kod znacznie utrudnia zmienianie testów.
5. I wreszcie nie przywiązuj się zbyt mocno do testów. Czasami najlepszym sposobem na przeprojektowanie czegoś jest po prostu usunięcie i rozpoczęcie od nowa.

Co powinieneś zobaczyć

Ćwiczenie 47. — sesja

```
$ nosetests
```

```
...
```

```
-----
Ran 3 tests in 0.008s
```

```
OK
```


To właśnie powinieneś zobaczyć, jeśli wszystko działa poprawnie. Spróbuj spowodować błąd, aby zobaczyć, jak to wygląda, a następnie napraw kod.

Zrób to sam

1. Poczytaj więcej o `nosetest`, a także o alternatywnych narzędziach.
2. Dowiedz się więcej o module `doctest` Pythona i sprawdź, czy bardziej Ci się podoba.
3. Ulepsz swój pokój, a następnie użyj go, aby ponownie zbudować grę, ale tym razem wykonuj testy jednostkowe na bieżąco.

Typowe pytania

Gdy uruchamiam `nosetests`, otrzymuję błąd składniowy. Jeśli otrzymasz taki błąd, przeanalizuj treść komunikatu i napraw te linie kodu, które są nad nim wyświetlone. Narzędzia, takie jak `nosetests`, uruchamiają i testują kod, więc znajdą błędy składniowe tak samo jak uruchomiony Python.

Nie mogę zaimportować `ex47.game`. Upewnij się, że utworzyłeś plik `ex47/__init__.py`. Wróć do ćwiczenia 46., aby zobaczyć, jak to się robi. Jeśli nie tu leży problem, w systemach macOS i Linux wykonaj następujące polecenie:

```
export PYTHONPATH=.
```

A w systemie Windows wpisz:

```
$env:PYTHONPATH = "$env:PYTHONPATH;."
```

Na koniec upewnij się, że uruchamiasz testy za pomocą `nosetests`, a nie przy użyciu samego Pythona.

Gdy uruchamiam `nosetest`, otrzymuję komunikat `UserWarning`. Prawdopodobnie masz zainstalowane dwie wersje Pythona lub nie korzystasz z `distribute`. Wróć i zainstaluj `distribute` lub `pip`, tak jak opisałem w ćwiczeniu 46.

Zaawansowane wprowadzanie danych przez użytkownika

W poprzednich grach obsługiwałeś wprowadzanie danych przez użytkownika, oczekując po prostu ustalonych łańcuchów znaków. Jeśli użytkownik wpisał „biegnę” i zrobił to dokładnie w ten sposób, gra działała. Jeśli wpisywał podobne zdania, na przykład „biegnę szybko”, gra nie działała. Potrzebujemy więc urządzenia, które pozwoli użytkownikom wpisywać frazy na różne sposoby, a następnie przekształci je w coś, co komputer rozumie. Chcielibyśmy na przykład, aby wszystkie poniższe zwroty działały tak samo:

- otwieram drzwi;
- otwórz drzwi;
- przechodzę PRZEZ drzwi;
- uderzam niedźwiedzia;
- Uderzam Niedźwiedzia w TWARZ.

Użytkownik powinien mieć możliwość wpisania w grze czegoś we własnym języku, a gra powinna się zorientować, co to znaczy. Napiszemy więc moduł, który właśnie to robi. W module będzie kilka klas, które współpracują ze sobą, aby obsłużyć dane wprowadzane przez użytkownika i przekształcić je w coś, co może zrobić Twoja gra.

Uproszczona wersja w języku angielskim mogłaby zawierać następujące elementy:

- słowa oddzielone spacjami;
- zdania złożone ze słów;
- gramatykę, która układa zdania w znaczenie.

Oznacza to, że najlepiej rozpocząć od wymyślenia, jak uzyskać słowa od użytkownika i jakie mają to być rodzaje słów.

Nasz leksykon gry

W naszej grze musimy utworzyć listę dozwolonych słów, którą nazwiemy „leksykonem”.

- Słowa kierunkowe: *north, south, east, west, down, up, left, right, back*.
- Czasowniki: *go, stop, kill, eat*.
- Słowa ze stop-listy: *the, in, of, from, at, it* i podobne.
- Rzeczowniki: *door, bear, princess, cabinet*.
- Liczby: dowolny ciąg znaków z zakresu od 0 do 9.

Z rzeczownikami mamy niewielki problem, ponieważ każdy pokój może mieć inny zestaw rzeczowników, ale na razie opracujemy ten mały zestaw, a ulepszymy go później.

Rozkładanie zdań na części

Gdy mamy już leksykon, potrzebujemy sposobu na rozbięcie zdań, abyśmy mogli dowiedzieć się, czym są. W naszym przypadku zdefiniowaliśmy zdanie jako „wyrazy rozdzielone spacjami”, więc musimy zrobić po prostu to:

```
stuff = input('> ')
words = stuff.split()
```

Na razie martwimy się tylko o to, ale ten sposób rozkładania zdań wystarczy nam na dość długi czas.

Krotki leksykonu

Kiedy już wiemy, jak podzielić zdanie na słowa, musimy po prostu przejrzeć listę słów i zorientować się, jakiego są „typu”. W tym celu wykorzystamy poręczną małą strukturę Pythona o nazwie „krotka” (ang. *tuple*). Krotka to lista, której nie można zmodyfikować. Powstaje po umieszczeniu w parze nawiasów danych rozdzielonych przecinkiem, podobnie jak lista.

```
first_word = ('verb', 'go')
second_word = ('direction', 'north')
third_word = ('direction', 'west')
sentence = [first_word, second_word, third_word]
```

W ten sposób tworzymy parę (*TYP*, *SŁOWO*), która pozwala przyjrzeć się danemu słowu i wykonać z nim różne czynności.

To tylko przykład, ale w zasadzie efekt końcowy. Chcemy przyjmować nieprzetworzone dane wejściowe od użytkownika, dzielić je na słowa za pomocą `split`, analizować te słowa w celu zidentyfikowania ich typów, a na końcu utworzyć z nich zdanie.

Skanowanie danych wejściowych

Teraz jesteś gotowy do napisania skanera. Ten skaner pobierze łańcuch znaków nieprzetworzonych danych wejściowych od użytkownika i zwróci zdanie, które składa się z listy krotek z parami (*TOKEN*, *SŁOWO*). Jeśli słowo nie jest częścią leksykonu, wtedy skaner powinien nadal zwracać *SŁOWO*, ale ustawić *TOKEN* na token błędu. Tokeny błędu będą informować użytkowników, że nabałaganili.

Tutaj zaczyna się zabawa. Nie powiem Ci, jak to zrobić. Zamiast tego napiszę „test jednostkowy”, a Ty napiszesz skaner, aby test jednostkowy zadziałał.

Wyjątki i liczby

Jest jedna drobna rzecz, z którą pomogę Ci z początku. To konwertowanie liczb. Aby to zrobić, będziemy oszukiwać i używać wyjątków. Wyjątkiem jest błąd otrzymywany z jakiejś funkcji, która mogła zostać uruchomiona. Funkcja „podnosi” wyjątek, kiedy napotka błąd, a Ty musisz obsłużyć ten wyjątek. Jeśli na przykład wpiszesz w Pythonie poniższy kod, otrzymasz wyjątek.

Ćwiczenie 48. — sesja Pythona

```
Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> int("hell")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hell'
```

`ValueError` jest wyjątkiem, który rzuciła funkcja `int()`, ponieważ to, co przekazałeś do `int()`, nie jest liczbą. Funkcja `int()` mogłaby zwrócić wartość informującą, że wystąpił błąd, ale ponieważ zwraca tylko liczby całkowite, miałyby z tym problem. Nie może zwrócić `-1`, ponieważ jest to liczba. Zamiast próbować dowiedzieć się, co zwrócić, gdy wystąpi błąd, funkcja `int()` podnosi wyjątek `ValueError`, a Ty go obsługujesz.

Wyjątki obsługuje się, używając słów kluczowych `try` i `except`.

ex48_convert.py

```
1 def convert_number(s):
2     try:
3         return int(s)
4     except ValueError:
5         return None
```

Kod, który chcesz „wypróbować”, wpisujesz wewnątrz bloku `try`, a następnie w bloku `except` umieszczasz kod do uruchomienia dla błędu. W tym przypadku chcemy „wypróbować” wywołanie `int()` na czymś, co może być liczbą. Jeśli daje błąd, „wyłapujemy” go i zwracamy `None`.

W pisany skanerze powinieneś użyć tej funkcji, aby sprawdzić, czy coś jest liczbą. Powinieneś również wykonać to jako ostatnią kontrolę przed zadeklarowaniem danego słowa jako słowa błędnego.

Wyzwanie „najpierw przygotuj testy”

„Najpierw przygotuj testy” to strategia programowania polegająca na tym, że najpierw piszesz zautomatyzowany test, który udaje, że kod działa, a *następnie* piszesz kod, aby test rzeczywiście zadziałał. Metoda sprawdza się wtedy, gdy nie możesz zwizualizować sobie sposobu zaimplementowania kodu, ale możesz sobie wyobrazić, jak musisz z nim pracować. Jeśli przykładowo wiesz, jak musisz użyć nowej klasy w innym module, ale nie wiesz jeszcze, jak ją zaimplementować — najpierw piszesz test.

Weźmiesz test, który Ci dam, i wykorzystasz go do napisania kodu powodującego, że test zadziała. Aby wykonać to ćwiczenie, postępuj zgodnie z poniższą procedurą.

1. Utwórz jedną małą część testu, który Ci dam.
2. Upewnij się, że ta część testu uruchamia się i *daje wynik negatywny*, żebyś wiedział, że test faktycznie potwierdza działanie funkcjonalności.

3. Przejdź do pliku źródłowego *lexicon.py* i napisz kod, który sprawia, że test daje wynik pozytywny.
4. Powtarzaj czynności, aż wszystko zostanie zaimplementowane w teście.

Gdy dojdiesz do kroku 3., warto połączyć to z inną metodą pisania kodu.

1. Utwórz „szkielet” funkcji lub klasę, której potrzebujesz.
2. Napisz wewnątrz komentarze opisujące działanie tej funkcji.
3. Napisz kod, który robi to, co opisują komentarze.
4. Usuń wszystkie komentarze, które po prostu powtarzają kod.

Tę metodę pisania kodu nazywamy „pseudokodem”; dobrze się sprawdza, kiedy nie wiesz, jak coś zaimplementować, ale możesz to opisać własnymi słowami.

Jeśli połączymy strategię „najpierw przygotuj testy” i „pseudokod”, otrzymamy prosty proces programowania.

1. Napisz trochę testu, który daje negatywny wynik.
2. Napisz szkielet funkcji, modułu lub klasy, których potrzebuje test.
3. Wypełnij szkielet komentarzami, opisując własnymi słowami, jak działa.
4. Zastępuj komentarze kodem, aż test zacznie dawać wynik pozytywny.
5. Powtórz czynności.

W tym ćwiczeniu będziesz wykorzystywał tę metodę pracy, sprawiając, żeby test, który Ci dam, zadziałał dla modułu *lexicon.py*.

Co powinieneś przetestować

Oto przypadek testowy *tests/lexicon_tests.py*, którego powinieneś użyć, ale *nie wpisuj go jeszcze*.

lexicon_tests.py

```
1  from nose.tools import *
2  from ex48 import lexicon
3
4
5  def test_directions():
6      assert_equal(lexicon.scan("north"), [('direction', 'north')])
7      result = lexicon.scan("north south east")
8      assert_equal(result, [('direction', 'north'),
9                             ('direction', 'south'),
10                            ('direction', 'east')])
11
12  def test_verbs():
13      assert_equal(lexicon.scan("go"), [('verb', 'go')])
14      result = lexicon.scan("go kill eat")
15      assert_equal(result, [('verb', 'go'),
```

```

16             ('verb', 'kill'),
17             ('verb', 'eat'))]]
18
19
20 def test_stops():
21     assert_equal(lexicon.scan("the"), [('stop', 'the')])
22     result = lexicon.scan("the in of")
23     assert_equal(result, [('stop', 'the'),
24                           ('stop', 'in'),
25                           ('stop', 'of')])
26
27
28 def test_nouns():
29     assert_equal(lexicon.scan("bear"), [('noun', 'bear')])
30     result = lexicon.scan("bear princess")
31     assert_equal(result, [('noun', 'bear'),
32                           ('noun', 'princess')])
33
34 def test_numbers():
35     assert_equal(lexicon.scan("1234"), [('number', 1234)])
36     result = lexicon.scan("3 91234")
37     assert_equal(result, [('number', 3),
38                           ('number', 91234)])
39
40
41 def test_errors():
42     assert_equal(lexicon.scan("ASDFADFASDF"),
43                 [('error', 'ASDFADFASDF')])
44     result = lexicon.scan("bear IAS princess")
45     assert_equal(result, [('noun', 'bear'),
46                           ('error', 'IAS'),
47                           ('noun', 'princess')])

```

Musisz zbudować nowy projekt za pomocą szkieletu projektu, podobnie jak w ćwiczeniu 47. Następnie będziesz musiał utworzyć ten przypadek testowy i plik *lexicon.py*, którego będzie używał. Spójrz na początek przypadku testowego, aby zobaczyć, jak jest importowany plik *lexicon.py*, i zorientować się, gdzie go umieścić.

Następnie postępuj zgodnie z procedurą, którą podałem, i pisz po trochu kod przypadku testowego. Przykładowo ja zrobiłbym to tak.

1. Napisz import u góry. Niech to zadziała.
2. Utwórz pustą wersję pierwszego przypadku testowego *test_directions*. Upewnij się, że działa.
3. Napisz pierwszą linię przypadku testowego *test_directions*. Niech daje wynik negatywny.
4. Przejdź do pliku *lexicon.py* i utwórz pustą funkcję *scan*.
5. Uruchom test i upewnij się, że funkcja *scan* jest w ogóle uruchamiana, nawet jeśli test się nie powiedzie.

6. Wypełnij komentarze w pseudokodzie, a dowiesz się, jak powinna działać funkcja `scan`, aby `test_directions` dał pozytywny wynik.
7. Pisz kod zgodny z komentarzami, aż test `test_directions` będzie dawał wynik pozytywny.
8. Wróć do `test_directions` i napisz pozostałe linie.
9. Wróć do funkcji `scan` w `lexicon.py` i popracuj nad nią, aby ten nowy kod testowy dawał pozytywny wynik.
10. Kiedy już to zrobisz, będziesz miał pierwszy zdawalny test i przejdziesz do następnego testu.

Dopóki będziesz przestrzegał tej zasady pisania po jednym małym kawałku na raz, możesz z powodzeniem zamieniać duży problem na mniejsze rozwiązywalne problemy. To jak wspinaczka na jedną wielką górę zamieniona na kilka podejść pod małe wzgórza.

Zrób to sam

1. Popraw test jednostkowy, aby przetestować więcej leksykonu.
2. Dodaj do leksykonu kolejne rzeczy, a następnie zaktualizuj test jednostkowy.
3. Upewnij się, że skaner obsługuje wprowadzanie przez użytkownika danych o dowolnej wielkości liter. Zaktualizuj test, aby upewnić się, że to faktycznie działa.
4. Znajdź inny sposób na konwersję liczb.
5. Moje rozwiązanie miało 37 linii. Czy Twoje jest dłuższe? A może krótsze?

Typowe pytania

Dlaczego ciągle otrzymuję `ImportErrors`? Błędy importu są zwykle spowodowane przez cztery rzeczy. 1. Nie utworzyłeś `__init__.py` w katalogu zawierającym moduły. 2. Jesteś w złym katalogu. 3. Importujesz niewłaściwy moduł, ponieważ źle napisałeś jego nazwę. 4. Twoja zmienna środowiskowa `PYTHONPATH` nie jest ustawiona na `.`, więc nie możesz ładować modułów z bieżącego katalogu.

Jaka jest różnica między `try-except` i `if-else`? Konstrukcja `try-except` służy tylko do obsługi wyjątków, które mogą być rzucane przez moduły. *Nigdy* nie powinna być używana jako alternatywa dla `if-else`.

Czy istnieje możliwość, aby gra toczyła się dalej, podczas gdy użytkownik wstrzymuje się z wpisaniem działania, które zamierza podjąć? Zakładam, że chcesz, aby użytkownika zaatakowały potwory, jeśli nie zareaguje wystarczająco szybko. Jest to możliwe, ale dotyczy modułów i technik, które leżą poza zakresem tej książki.

Tworzenie zdań

Z naszego małego skanera leksykonu gry powinniśmy być w stanie uzyskać listę, która wygląda następująco.

Ćwiczenie 49. — sesja Pythona

```
Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from ex48 import lexicon
>>> lexicon.scan("go north")
[('verb', 'go'), ('direction', 'north')]
>>> lexicon.scan("kill the princess")
[('verb', 'kill'), ('stop', 'the'), ('noun', 'princess')]
>>> lexicon.scan("eat the bear")
[('verb', 'eat'), ('stop', 'the'), ('noun', 'bear')]
```

Będzie to również działać na dłuższych zdaniach, takich jak `lexicon.scan("open the door and smack the bear in the nose")`.

Teraz zmienimy to w coś, z czym gra może pracować, a może to być coś w rodzaju klasy `Sentence`. Jeśli pamiętasz ze szkoły podstawowej, zdanie może być prostą strukturą, taką jak:

Podmiot orzeczenie dopełnienie.

Oczywiście jest to bardziej skomplikowane i prawdopodobnie wiele dni poświęciłeś na rozwiązywanie denerwujących schematów budowy zdań na zajęciach z języka polskiego. Chcemy przekształcić powyższą listę krotek w ładny obiekt `Sentence`, który ma podmiot (`subject`), orzeczenie (`verb`) i dopełnienie (`object`).

Dopasowywanie i podglądanie

W tym celu potrzebujemy pięciu narzędzi. Oto one.

1. Sposób na zapętlanie przez listę zeskanowanych słów. To łatwe.
2. Sposób na „dopasowywanie” różnych typów krotek, których oczekujemy w naszej konfiguracji „podmiot orzeczenie dopełnienie”.
3. Sposób na „podejrzenie” potencjalnej krotki, abyśmy mogli podjąć pewne decyzje.
4. Sposób na „pomijanie” rzeczy, na których nam nie zależy, takich jak słowa ze `stop-listy`.
5. Obiekt `Sentence`, w którym umieścimy wyniki.

Będziemy umieszczać te funkcje w module o nazwie `ex48.parser`, w pliku o nazwie `ex48/parser.py`, aby je przetestować. Użyjemy funkcji `peek`, aby powiedzieć: „Spójrz na następny element na naszej liście krotek, a następnie dopasuj, aby go zdjąć z listy i pracować z nim”.

Gramatyka zdania

Zanim będziesz mógł napisać kod, musisz zrozumieć, jak działa podstawowa gramatyka. W naszym parserze chcemy wygenerować obiekt `Sentence`, który ma trzy atrybuty.

Sentence.subject — jest to podmiot każdego zdania, ale w większości przypadków domyślnie może to być „gracz”, ponieważ zdanie: „Biegnę na północ” oznacza: „Gracz biegnie na północ”. To będzie rzeczownik.

Sentence.verb — to jest działanie podejmowane w zdaniu przez podmiot. W „Gracz biegnie na północ” jest to „biegnie”. To będzie czasownik.

Sentence.object — jest to kolejny rzeczownik, który odnosi się do tego, na czym wykonywana jest czynność. W naszej grze rozdzielamy kierunki, które również będą dopełnieniami. W „Gracz biegnie na północ” dopełnieniem będzie słowo „północ”. W „Gracz uderza niedźwiedzia” dopełnieniem będzie „niedźwiedź”.

Nasz parser musi korzystać z funkcji, które opisaliśmy, i każde zeskanowane zdanie konwertować na listę obiektów `Sentence`, aby dopasować dane wejściowe.

Słowo o wyjątkach

Dowiedziałeś się w skrócie, czym są wyjątki, ale nie wiesz, jak je podnosić. Kod z klasą `ParserError` na początku demonstruje, jak to zrobić. Zwróć uwagę, że kod używa klas, aby przekazać typ wyjątku (`Exception`). Zwróć również uwagę użycie słowa kluczowego `raise` w celu podnoszenia wyjątków.

W Twoich testach będziesz pracował z tymi wyjątkami i pokażę Ci, jak je zrobić.

Kod parsera

Jeśli chcesz dodatkowego wyzwania, zatrzymaj się teraz i spróbuj napisać to jedynie na podstawie mojego opisu. Jeśli na czymś utkniesz, możesz wrócić i zobaczyć, jak to zrobiłem, ale próba samodzielnej implementacji parsera jest dobrym ćwiczeniem. Przeprowadzę Cię teraz przez kod, abyś mógł wpisać go do swojego pliku `ex48/parser.py`. Zaczynamy parser od wyjątku, którego potrzebujemy dla błędu parsowania.

parser.py

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

W ten sposób tworzysz własną klasę `ParserError` dla wyjątków, które możesz rzucać. Następnie potrzebujemy obiektu `Sentence`, który utworzymy.

parser.py

```
1 class Sentence(object):
2
3     def __init__(self, subject, verb, obj):
4         # pamiętaj, że bierzemy krotki ('noun','princess') i konwertujemy je
5         self.subject = subject[1]
6         self.verb = verb[1]
7         self.object = obj[1]
```

Na razie w tym kodzie nie ma nic szczególnego. Po prostu tworzysz prostą klasę.

W naszym opisie problemu potrzebowaliśmy funkcji, która może zaglądać (ang. *peek*) do listy słów i zwracać, jaki to typ słowa.

parser.py

```
1 def peek(word_list):
2     if word_list:
3         word = word_list[0]
4         return word[0]
5     else:
6         return None
```

Potrzebujemy tej funkcji, ponieważ na podstawie tego, jakie jest następne słowo, będziemy musieli podejmować decyzje dotyczące rodzaju zdania, z którym mamy do czynienia. Następnie możemy wywołać inną funkcję, aby skonsumować to słowo i kontynuować.

Aby skonsumować słowo, używamy funkcji *match*, która potwierdza, że oczekiwane słowo jest właściwym typem, usuwa je z listy i zwraca to słowo.

parser.py

```
1 def match(word_list, expecting):
2     if word_list:
3         word = word_list.pop(0)
4
5         if word[0] == expecting:
6             return word
7         else:
8             return None
9     else:
10        return None
```

Tym razem również jest to dość proste, ale upewnij się, że rozumiesz kod. Upewnij się także, że rozumiesz, *dlaczego* robię to w ten sposób. Muszę zerknąć na słowa na liście, aby zdecydować, z jakim rodzajem zdania mam do czynienia, a następnie muszę dopasować te słowa, aby utworzyć zdanie (obiekt *Sentence*).

Ostatnią rzeczą, której potrzebuję, jest sposób na pomijanie (ang. *skip*) słów, które nie są użyteczne w obiekcie *Sentence*. To są słowa oznaczone jako „stop-lista” (wpisujesz 'stop'), którymi są spójniki, przyimki i zaimki, na przykład „i”, „lub”, „na”, „pod”, „ta” i tak dalej.

parser.py

```
1 def skip(word_list, word_type):
2     while peek(word_list) == word_type:
3         match(word_list, word_type)
```

Pamiętaj, że funkcja `skip` nie pomija jednego słowa — pomija tyle słów tego samego typu, ile znajdzie. Tak więc jeśli ktoś wpisze „Krzyczę na tego niedźwiedzia”, dostaniesz „krzyczę” i „niedźwiedzia”.

To jest nasz podstawowy zestaw funkcji parsowania, dzięki któremu możemy parsować praktycznie każdy tekst, który chcemy. Nasz parser jest jednak bardzo prosty, więc pozostałe funkcje są krótkie.

Najpierw możemy obsłużyć parsowanie orzeczenia (czasownika).

parser.py

```
1 def parse_verb(word_list):
2     skip(word_list, 'stop')
3
4     if peek(word_list) == 'verb':
5         return match(word_list, 'verb')
6     else:
7         raise ParserError("Następnie oczekiwany jest czasownik.")
```

Pomijamy wszystkie słowa ze stop-listy, a następnie zaglądamy, czy następne słowo jest typu `verb`. Jeśli tak nie jest, podnosimy `ParserError`, aby powiedzieć dlaczego. Jeśli jest to `verb`, dopasowujemy je, co spowoduje usunięcie z listy. Podobna funkcja obsługuje obiekty zdań.

parser.py

```
1 def parse_object(word_list):
2     skip(word_list, 'stop')
3     next_word = peek(word_list)
4
5     if next_word == 'noun':
6         return match(word_list, 'noun')
7     elif next_word == 'direction':
8         return match(word_list, 'direction')
9     else:
10        raise ParserError("Następnie oczekiwany jest rzeczownik lub kierunek.")
```

Ponownie pomijamy słowa ze stop-listy, zaglądamy i decydujemy, czy zdanie jest poprawne, na podstawie tego, co tam jest. Jednak w funkcji `parse_object` musimy obsłużyć słowa typu „rzeczownik”, jak i typu „kierunek” jako możliwe dopełnienia. Następnie podmioty są znowu podobne, ale ponieważ chcemy obsłużyć domyślny rzeczownik „gracz”, musimy użyć `peak`.

parser.py

```
1 def parse_subject(word_list):
2     skip(word_list, 'stop')
3     next_word = peek(word_list)
```

```

4
5     if next_word == 'noun':
6         return match(word_list, 'noun')
7     elif next_word == 'verb':
8         return ('noun', 'player')
9     else:
10        raise ParserError("Następnie oczekiwany jest czasownik.")

```

Gdy mamy już to wszystko gotowe, nasza ostatnia funkcja `parse_sentence` jest bardzo prosta.

parser.py

```

1  def parse_sentence(word_list):
2      subj = parse_subject(word_list)
3      verb = parse_verb(word_list)
4      obj = parse_object(word_list)
5
6      return Sentence(subj, verb, obj)

```

Zabawa z parserem

Aby zobaczyć, jak to działa, możesz pobawić się z parserem w ten sposób.

Ćwiczenie 49a — sesja Pythona

```

Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from ex48.parser import *
>>> x = parse_sentence([('verb', 'run'), ('direction', 'north')])
>>> x.subject
'player'
>>> x.verb
'run'
>>> x.object
'north'
>>> x = parse_sentence([('noun', 'bear'), ('verb', 'eat'), ('stop', 'the'),
...                     ('noun', 'honey')])
>>> x.subject
'bear'
>>> x.verb
'eat'
>>> x.object
'honey'

```

Spróbuj odwzorować zdania na prawidłowe dopasowania w zdaniu. Jak na przykład powiedziałyby: „Ten niedźwiedź biegnie na południe”?

Co powinienesz przetestować

W ćwiczeniu 49. napisz kompletny test, który potwierdza, że wszystko w tym kodzie działa. Umieść ten test w katalogu `tests/parser_tests.py`, podobnie jak w przypadku pliku testowego z poprzedniego ćwiczenia. Test obejmuje generowanie powstawania wyjątków przez podawanie parserowi złych zdań.

Wykonaj sprawdzanie pod kątem wyjątku, używając funkcji `assert_raises` z dokumentacji `nose`. Naucz się, jak tego używać, abyś mógł napisać test, który *ma się nie powieść*, co jest bardzo ważne w testowaniu. Poczytaj o tej funkcji (i innych) w dokumentacji `nose`.

Kiedy skończysz, powinienesz wiedzieć, jak działa ten kawałek kodu i jak napisać test dla cudzego kodu, nawet jeśli dana osoba nie chce tego od Ciebie. Zaufaj mi, to bardzo przydatna umiejętność.

Zrób to sam

1. Zmień metody `parse_` i spróbuj umieścić je w klasie, zamiast używać ich po prostu jako metod. Które podejście bardziej Ci się podoba?
2. Spraw, by parser był bardziej odporny na błędy, aby uniknąć irytowania użytkowników, gdy wpisują słowa, których Twój leksykon nie rozumie.
3. Popraw gramatykę, dodając obsługę większej liczby rzeczy, takich jak liczby.
4. Zastanów się, w jaki sposób możesz użyć klasy `Sentence` w swojej grze, aby robić więcej ciekawych rzeczy z danymi wpisywanymi przez użytkownika.

Typowe pytania

Nie mogę sprawić, żeby funkcja `assert_raises` zadziałała poprawnie. Upewnij się, że wpisujesz `assert_raises(exception, callable, parameters)`, a *nie* `assert_raises(exception, callable(parameters))`. Zwróć uwagę, w jaki sposób ta druga forma wywołuje funkcję, a następnie przekazuje wynik do `assert_raises`, co jest nieprawidłowe. Musisz przekazać funkcję do wywołania, *natomiast* jej argumenty do `assert_raises`.

Twoja pierwsza strona internetowa

Trzy ostatnie ćwiczenia będą bardzo trudne i nie spiesz się z nimi. W pierwszym zbudujesz prostą internetową wersję jednej z Twoich gier. Przed wykonaniem tego ćwiczenia *musisz* pomyślnie ukończyć ćwiczenie 46. i mieć działające zainstalowane narzędzie `pip`, żebyś mógł instalować pakiety. Musisz także wiedzieć, jak utworzyć katalog szkieletu projektu. Jeśli nie pamiętasz, jak to zrobić, wróć do ćwiczenia 46. i zrób je od nowa.

Instalowanie frameworku flask

Zanim utworzysz swoją pierwszą aplikację internetową, musisz najpierw zainstalować „framework webowy” zwany flask. Termin „framework” oznacza — ogólnie rzecz biorąc — „jakiś pakiet, który ułatwia wykonywanie pewnych czynności”. W świecie aplikacji internetowych ludzie tworzą „frameworki webowe”, aby zrekompensować sobie trudne problemy, jakie napotkali podczas tworzenia własnych witryn. Dzielą się tymi rozwiązaniami. Można je pobrać w postaci pakietu, aby załadować do własnych projektów.

W naszym przypadku użyjemy frameworku flask, ale jest wiele, wiele, *wiele innych*, z których możesz wybierać. Na razie naucz się frameworku flask, a następnie przejdź do innych, gdy będziesz już gotowy (lub po prostu dalej używaj flask, ponieważ jest wystarczająco dobry).

Użyj narzędzia `pip`, aby zainstalować framework flask.

```
$ sudo pip install flask
[sudo] password for zedshaw:
Downloading/unpacking flask
  Running setup.py egg_info for package flask

Installing collected packages: flask
  Running setup.py install for flask

Successfully installed flask
Cleaning up...
```

Polecenie będzie działać na komputerach z systemami Linux i macOS, ale w systemie Windows po prostu opuść fragment `sudo` polecenia `pip install` i powinno działać. Jeśli nie, wróć do ćwiczenia 46. i upewnij się, że potrafisz się tym poleceniem niezawodnie posługiwać.

Tworzenie prostego projektu „Witaj, świecie”

Teraz przygotujesz bardzo prostą aplikację internetową „Witaj, świecie” i katalog projektu, używając frameworku flask. Najpierw utwórz katalog projektu.

```
$ cd projects
$ mkdir gothonweb
$ cd gothonweb
$ mkdir bin gothonweb tests docs templates
$ touch gothonweb/__init__.py
$ touch tests/__init__.py
```

Weźmiesz grę z ćwiczenia 43. i przekształcisz ją w aplikację internetową, dlatego przyjąłem nazwę *gothonweb*. Zanim to zrobisz, musisz utworzyć najbardziej podstawową aplikację flask. Umieść następujący kod w pliku *app.py*.

ex50.py

```
1  from flask import Flask
2  app = Flask(__name__)
3
4  @app.route('/')
5  def hello_world():
6      return 'Witaj, świecie!'
7
8  if __name__ == "__main__":
9      app.run()
```

Uruchom tę aplikację w następujący sposób.

```
(lpthw) $ python3.6 app.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Wreszcie użyj przeglądarki internetowej i przejdź do adresu *http://localhost:5000/*, a powinieneś zobaczyć dwie rzeczy. Po pierwsze, w przeglądarce zobaczysz komunikat *Witaj, świecie!*. Po drugie, zobaczysz terminal z nowymi danymi wyjściowymi, takimi jak te:

```
(lpthw) $ python3.6 app.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [22/Feb/2017 14:28:50] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [22/Feb/2017 14:28:50] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [22/Feb/2017 14:28:50] "GET /favicon.ico HTTP/1.1" 404 -
```

Są to komunikaty dziennika, które wyświetla flask, dzięki czemu można zobaczyć, że serwer działa i co robi przeglądarka za kulisami. Komunikaty dziennika pomagają w debugowaniu, kiedy masz problemy. Komunikat mówi na przykład, że Twoja przeglądarka próbowała pobrać plik */favicon.ico*, ale ten plik nie istnieje, więc zwróciła kod statusu *404 Not Found*.

Nie wyjaśniłem jeszcze, jak działa *jakakolwiek* z tych internetowych funkcji, ponieważ chcę, abyś był przygotowany na to, gdy zrobię to szczegółowo w następnych dwóch ćwiczeniach. Aby to osiągnąć, poproszę Cię o popsucie Twojej aplikacji flask na różne sposoby, a następnie zrestrukturyzowanie jej, żebyś wiedział, jak jest skonfigurowana.

Co się tutaj dzieje

Oto, co dzieje się, gdy przeglądarka natrafi na Twoją aplikację.

1. Twoja przeglądarka nawiązuje połączenie sieciowe z Twoim własnym komputerem, który nazywa się *localhost* i jest standardowym sposobem powiedzenia: „Jakaś tam nazwa, po której mój komputer jest rozpoznawany w sieci”. Przeglądarka używa również portu 5000.
2. Po nawiązaniu połączenia przeglądarka wysyła żądanie HTTP do aplikacji *app.py* i prosi o podanie adresu URL `/`, który jest zwykle pierwszym adresem URL w każdej witrynie.
3. Wewnątrz aplikacji *app.py* masz listę adresów URL wraz z funkcjami, którym odpowiadają. Jedyną, którą mamy, to `/`, czyli mapowanie na `'index'`. Oznacza to, że ilekroć ktoś przejdzie do adresu `/` za pomocą przeglądarki, flask znajdzie `def index` i uruchomi to, aby obsłużyć żądanie.
4. Gdy flask znajdzie funkcję `def index`, wywołuje ją, aby faktycznie obsłużyć żądanie. Funkcja ta uruchamia się i po prostu zwraca łańcuch znaków dla tego, co flask powinien wysłać do przeglądarki.
5. Wreszcie flask obsługuje żądanie i wysyła tę odpowiedź do przeglądarki; to właśnie widzisz.

Upewnij się, że naprawdę wszystko rozumiesz. Sporządź diagram opisujący sposób, w jaki informacje te płyną z przeglądarki do flask, następnie do `def index` i z powrotem do przeglądarki.

Naprawianie błędów

Najpierw usuń linię 8., w której przypisujesz zmienną `greeting`, a następnie odśwież stronę w przeglądarce. Potem użyj `Ctrl+C`, aby zamknąć flask i zacząć od nowa. Po ponownym uruchomieniu odśwież przeglądarkę i powinieneś zobaczyć komunikat *Internal Server Error*. Po powrocie do terminalu zobaczysz coś takiego ([*VENV*] jest ścieżką do Twojego katalogu `.venvs/`):

```
(lpthw) $ python3.6 app.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
[2017-02-22 14:35:54,256] ERROR in app: Exception on / [GET]
Traceback (most recent call last):
  File "[VENV]/site-packages/flask/app.py",
    line 1982, in wsgi_app
    response = self.full_dispatch_request()
  File "[VENV]/site-packages/flask/app.py",
    line 1614, in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "[VENV]/site-packages/flask/app.py",
    line 1517, in handle_user_exception
    reraise(exc_type, exc_value, tb)
  File "[VENV]/site-packages/flask/_compat.py",
```



```
line 33, in reraise
    raise value
File "[VENV]/site-packages/flask/app.py",
line 1612, in full_dispatch_request
    rv = self.dispatch_request()
File "[VENV]/site-packages/flask/app.py",
line 1598, in dispatch_request
    return self.view_functions[rule.endpoint](**req.view_args)
File "app.py", line 8, in index
    return render_template("index.html", greeting=greeting)
NameError: name 'greeting' is not defined
127.0.0.1 - - [22/Feb/2017 14:35:54] "GET / HTTP/1.1" 500 -
```

Działa to dość dobrze, ale możesz także uruchomić flask w „trybie debuggera”. Dostaniesz lepszą stronę błędów i więcej przydatnych informacji. Problem z trybem debuggera polega na tym, że nie można go bezpiecznie uruchamiać w internecie, więc musisz go bezpośrednio włączyć w ten sposób:

```
(lpthw) $ export FLASK_DEBUG=1
(lpthw) $ python3.6 app.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 222-752-342
```

Odśwież przeglądarkę, a otrzymasz o wiele bardziej szczegółową stronę z informacjami, których może użyć do debugowania aplikacji, oraz z konsolą do pracy „na żywo”, aby dowiedzieć się więcej.

OSTRZEŻENIE! To właśnie konsola do debugowania na żywo i ulepszone dane wyjściowe frameworku flask powodują, że tryb debuggera jest tak niebezpieczny w internecie. Dzięki tym informacjom atakujący może całkowicie zdalnie sterować Twoją maszyną. Jeśli kiedykolwiek umieścisz swoją aplikację internetową w internecie, *nie aktywuj* trybu debuggera. W rzeczywistości lepiej byłoby uniknąć tak łatwego aktywowania FLASK_DEBUG. Kuszące jest to, by po prostu zhakować ten startup i zaoszczędzić sobie tego kroku w trakcie rozwoju aplikacji, ale wtedy ten hack wejdzie na Twój serwer internetowy i zmieni się w prawdziwy hack, a nie tylko coś, co zrobiłeś z lenistwa pewnej nocy, kiedy byłeś zmęczony.

Tworzenie podstawowych szablonów

Możesz popsuć swoją aplikację flask, ale czy zauważyłeś, że *Witaj, świecie* nie jest zbyt dobrą stroną HTML? Jest to aplikacja internetowa i jako taka potrzebuje prawidłowej odpowiedzi HTML. Aby to zrobić, utworzysz prosty szablon z napisem „Witaj, świecie” wyświetlonym dużą zieloną czcionką.

Pierwszym krokiem jest utworzenie pliku *templates/index.html*.

index.html

```

<html>
  <head>
    <title>Goci z planety Percal 25</title>
  </head>
  <body>

    {% if greeting %}
      Chciałem tylko powiedzieć
      <em style="color: green; font-size: 2em;">{{ greeting }}</em>.
    {% else %}
      <em>Witaj</em>, świecie!
    {% endif %}

  </body>
</html>

```

Jeśli wiesz, czym jest HTML, powinno to wyglądać dość znajomo. Jeśli nie, poszukaj informacji o HTML-u i spróbuj napisać kilka stron internetowych ręcznie, żebyś wiedział, jak to działa. Ten plik HTML jest jednak *szablonem*, co oznacza, że flask wypełni „dziury” w tekście w zależności od zmiennych, które wprowadzisz do szablonu. Każde miejsce, gdzie widzisz \$greeting, będzie zmienną, którą przekażesz do szablonu zmieniającego jej zawartość.

Aby Twoja aplikacja *app.py* to zrobiła, musisz dodać kod informujący flask, gdzie umieścić szablon i go zrenderować. Zmień ten plik w następujący sposób.

app.py

```

1  from flask import Flask
2  from flask import render_template
3
4  app = Flask(__name__)
5
6  @app.route("/")
7  def index():
8      greeting = "Witaj, świecie"
9      return render_template("index.html", greeting=greeting)
10
11  if __name__ == "__main__":
12      app.run()

```

Zwróć szczególną uwagę na nową zmienną `render` i sposób zmiany ostatniej linii `index.GET`, żeby zwracała `render.index()`, przekazując zmienną `greeting`.

Gdy już to zrobisz, załaduj ponownie stronę internetową w przeglądarce. Powinieneś zobaczyć inny komunikat na zielono. Powinieneś być w stanie również podejrzeć źródło strony w przeglądarce, aby zobaczyć, że jest to prawidłowy HTML.

To było dość szybkie tłumaczenie, więc pozwól, że wyjaśnię, jak działa szablon.

1. W Twoim pliku *app.py* zaimportowałeś u góry nową funkcję o nazwie `render_template`.
2. Funkcja `render_template` „wie”, jak ładować pliki *.html* z katalogu *templates/*, ponieważ jest to domyślne ustawienie magiczne dla aplikacji *flask*.
3. W dalszej części kodu, gdy przeglądarka trafi na `def index`, zamiast po prostu zwrócić łańcuch znaków `greeting`, wywołujesz `render_template` i przekazujesz do niej `greeting` jako zmienną.
4. Metoda `render_template` ładuje następnie plik *templates/index.html* (nawet jeśli nie powiedziałeś wyraźnie *templates*) i przetwarza go.
5. W pliku *templates/index.html* masz coś, co wygląda jak normalny HTML, ale potem jest „kod” umieszczony między dwoma rodzajami znaczników. Jednym z nich jest `{% %}`, który oznacza fragmenty „wykonywalnego kodu” (instrukcje `if`, pętle `for` i podobne). Drugi to `{{ }}`, który oznacza zmienne do przekonwertowania na tekst i umieszczenia w wyjściu HTML. Kod wykonywalny `{% %}` nie pojawia się w HTML-u. Gdy chcesz dowiedzieć się więcej o tym języku szablonów, przeczytaj dokumentację Jinja2.

Aby się w to zagłębić, zmień zmienną `greeting` i kod HTML, aby zobaczyć, jaki to da efekt. Utwórz także inny szablon o nazwie *templates/foo.html* i zrenderuj go, podobnie jak wcześniej.

Zrób to sam

1. Przeczytaj dokumentację na stronie <http://flask.pocoo.org/docs/0.12/>, która jest tym samym, co projekt *flask*.
2. Poeksperymentuj ze wszystkim, co możesz tam znaleźć, łącznie z przykładowym kodem.
3. Poczytaj o HTML5 i CSS3. Utwórz kilka innych plików *.html* i *.css*, żeby poćwiczyć.
4. Jeśli masz znajomego, który zna Django i chętnie Ci pomoże, rozważ wykonanie ćwiczeń 50., 51. i 52. w Django, aby zobaczyć, jak to jest.

Typowe pytania

Nie mogę połączyć się z *http://localhost:5000/*. Spróbuj zamiast tego *http://127.0.0.1:5000/*.

Jaka jest różnica między *flask* i *web.py*? Nie ma różnicy. Po prostu „zablokowałem” *web.py* na konkretną wersję, aby było to spójne dla uczniów, a następnie nazwałem to *flask*. Późniejsze wersje *web.py* mogą się różnić od tej wersji.

Nie mogę znaleźć *index.html* (ani niczego innego). Prawdopodobnie robisz najpierw `cd bin/`, a potem próbujesz pracować z projektem. Nie rób tego. Wszystkie polecenia i instrukcje zakładają, że jesteś w katalogu o jeden poziom wyżej niż *bin/*, więc jeśli nie możesz wpisać `python3.6 app.py`, to jesteś w złym katalogu.

Dlaczego przypisujemy `greeting=greeting`, gdy wywołujemy szablon? Nie przypisujesz do `greeting`. Ustawiasz nazwany parametr do podania szablonowi. To coś w rodzaju przypisania, ale wpływa tylko na wywołanie funkcji szablonu.

Nie mogę używać portu 5000 na moim komputerze. Prawdopodobnie masz zainstalowany program antywirusowy, który używa tego portu. Spróbuj innego portu.

Po zainstalowaniu flask otrzymuję `ImportError "No module named web"`. Najprawdopodobniej masz zainstalowanych wiele wersji Pythona i używasz niewłaściwej lub nie wykonałeś poprawnie instalacji z powodu starej wersji `pip`. Spróbuj usunąć instalację `flask` i zainstalować go ponownie. Jeśli to nie zadziała, upewnij się kilka razy, że używasz właściwej wersji Pythona.

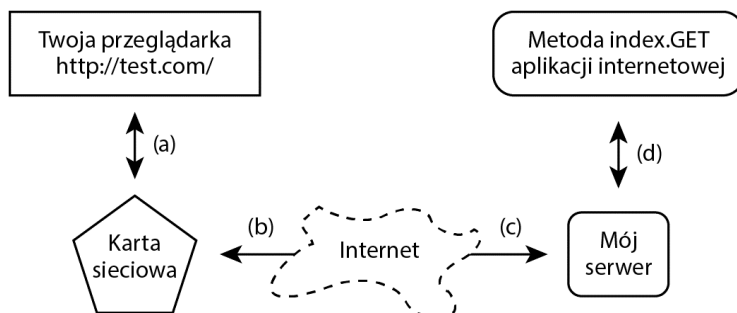
Pobieranie danych wejściowych z przeglądarki

Choć ekscytujące jest to, gdy przeglądarka wyświetla komunikat *Witaj, świecie*, jeszcze bardziej ekscytujące jest umożliwienie użytkownikowi wysyłania tekstu do aplikacji przy użyciu formularza. W tym ćwiczeniu poprawimy naszą początkową aplikację internetową; dodamy formularze i przechowywanie informacji o użytkownikach w ich „sesjach”.

Jak działa sieć

Czas na nudne rzeczy. Zanim będziesz mógł tworzyć formularze, musisz zrozumieć podstawowe zasady działania sieci. Ten opis nie jest kompletny, ale jest dokładny i pomoże zrozumieć, co złego może zdarzyć się w Twojej aplikacji. Tworzenie formularzy również będzie łatwiejsze, jeśli dowiesz się, co robią.

Zacznę od prostego schematu, który pokazuje różne części żądania internetowego oraz sposób przepływu informacji.



Oznaczyłem strzałki literami, żebym mógł krok po kroku przeprowadzić Cię przez proces standardowego żądania.

1. Wpisujesz w przeglądarce adres URL *http://test.com/*, a przeglądarka wysyła na linii (a) żądanie do karty sieciowej Twojego komputera.
2. Twoje żądanie zostaje wysłane przez internet na linii (b), a następnie jest przekazywane do zdalnego komputera na linii (c), gdzie *Mój serwer* akceptuje żądanie.
3. Gdy *Mój serwer* je zaakceptuje, moja aplikacja internetowa otrzymuje je na linii (d), a mój kod Pythona uruchamia procedurę obsługi `index.GET`.
4. Gdy *zwracam odpowiedź*, wychodzi ona z mojego serwera Pythona i kieruje się do Twojej przeglądarki ponownie przez linię (d).

5. Fizyczny serwer, na którym działa ta strona, otrzymuje odpowiedź na linii (d), a następnie odsyła ją z powrotem przez internet na linii (c).
6. Następnie odpowiedź z serwera przychodzi z internetu na linii (b), a karta sieciowa Twojego komputera przekazuje ją do przeglądarki na linii (a).
7. Na koniec przeglądarka wyświetla odpowiedź.

W tym opisie jest kilka terminów, które powinienś znać, żebyśmy korzystali ze wspólnego słownictwa, gdy będziemy rozmawiać o Twojej aplikacji internetowej.

Przeglądarka. Oprogramowanie, z którego prawdopodobnie korzystasz każdego dnia. Większość ludzi nie wie, co przeglądarka naprawdę robi. Nazywają przeglądarkę po prostu „internetem”. Jej zadaniem jest przyjmowanie adresów (na przykład *http://test.com/*), które wpisujesz w pasku adresowym, a następnie używanie tych informacji do wysyłania żądań do serwera znajdującego się pod wskazanym adresem.

Adres. Jest to zwykle adres URL (ang. *Uniform Resource Locator*), taki jak *http://test.com/*, który wskazuje, gdzie powinna kierować się przeglądarka. Pierwsza część, *http*, wskazuje protokół, którego chcesz użyć, w tym przypadku *Hyper-Text Transport Protocol*. Możesz również spróbować wpisać *ftp://ftp.ibiblio.org/*, aby zobaczyć, jak działa File Transport Protocol. Część *http://test.com/* to „nazwa hosta”, adres czytelny dla człowieka, który możesz zapamiętać i który mapuje się na numer zwany adresem IP — podobny do numeru telefonu, ale dla komputera w internecie. I wreszcie adresy URL mogą mieć końcową ścieżkę, taką jak część */book/* adresu *http://test.com/book/*, która wskazuje plik lub jakieś zasoby na serwerze do pobrania za pomocą żądania. Istnieje jeszcze wiele innych części, ale te są najważniejsze.

Połączenie. Kiedy przeglądarka „wie”, jakiego protokołu chcesz użyć (*http*), z którym serwerem chcesz się komunikować (*http://test.com/*) i jakie zasoby z tego serwera pobrać, musi nawiązać połączenie. Przeglądarka prosi po prostu system operacyjny (OS), aby otworzył „port” na komputerze, zwykle port 80. Kiedy to się stanie, system operacyjny przekazuje programowi coś, co działa jak plik, ale w rzeczywistości wysyła i odbiera bajty przez przewody sieciowe pomiędzy Twoim komputerem i drugim komputerem pod adresem *http://test.com/*. To samo dzieje się z *http://localhost:8080/*, ale w tym przypadku mówisz przeglądarce, aby połączyła się z Twoim własnym komputerem (*localhost*) i użyła portu 8080 zamiast domyślnego 80. Możesz także wpisać *http://test.com:80/* i uzyskać taki sam wynik, tyle że wyraźnie mówisz, że chcesz używać portu 80, zamiast polegać na wartości domyślnej.

Żądanie. Twoja przeglądarka jest połączona przy użyciu podanego adresu. Teraz musi poprosić o wymagany (przez Ciebie) zasób znajdujący się na zdalnym serwerze. Jeśli podałeś */book/* na końcu adresu URL, wtedy potrzebujesz pliku (zasobu) znajdującego się w */book/*, a większość serwerów użyje prawdziwego pliku */book/index.html*, ale będzie udawać, że on nie istnieje. Aby uzyskać ten zasób, przeglądarka wysyła do serwera żądanie. Nie będę opisywał dokładnie, jak to działa, ale ważne jest, że musi wysłać coś, aby zapytać serwer o to żądanie. Ciekawe jest, że te „zasoby” nie muszą być plikami. Gdy na przykład przeglądarka w Twojej aplikacji pyta o coś, serwer zwraca coś, co wygenerował Twój kod Pythona.

Serwer. Jest to komputer na końcu połączenia przeglądarki, który odpowiada na żądania przeglądarki dotyczące plików (zasobów). Większość serwerów WWW wysyła po prostu pliki, to w rzeczywistości większość ruchu. Jednak Ty zbudujesz w Pythonie serwer, który będzie wiedział, jak przyjmować żądania o zasoby, a następnie będzie zwracał łańcuchy znaków, które przygotowałeś przy użyciu Pythona. Kiedy przygotujesz te łańcuchy znaków, udajesz, że jesteś plikiem dla przeglądarki, ale naprawdę będzie to tylko kod. Jak widziałeś w ćwiczeniu 50., przygotowanie odpowiedzi również nie wymaga dużo kodu.

Odpowiedź. Jest to kod HTML (CSS, JavaScript lub obrazy), który Twój serwer chce odesłać do przeglądarki jako odpowiedź na jej żądanie. W przypadku plików serwer po prostu odczytuje je z dysku i wysyła do przeglądarki, ale opakowuje zawartość dysku w specjalny „nagłówek”, więc przeglądarka wie, co otrzyma. W przypadku Twojej aplikacji wysyłasz te same rzeczy, w tym nagłówki, ale generujesz te dane w locie za pomocą kodu Pythona.

To najszybszy kurs dotyczący tego, jak przeglądarka internetowa uzyskuje dostęp do informacji na serwerach w internecie. Powinno Ci to wystarczyć do zrozumienia tego ćwiczenia, ale jeśli tak nie jest, poczytaj więcej, dopóki tego nie przyswoisz. Naprawdę dobrym sposobem jest zrobienie schematu i rozbicie różnych części aplikacji internetowej, którą utworzyłeś w ćwiczeniu 50. Jeśli będziesz umiał za pomocą schematu rozłożyć na części Twoją aplikację internetową w przewidywalny sposób, zaczniesz rozumieć, jak działa.

Jak działają formularze

Najlepszym sposobem, żeby pobawić się formularzami, jest napisanie kodu akceptującego dane formularza, a następnie zobaczenie, co można z tym zrobić. Otwórz plik *app.py* i wstaw do niego następujący kod.

form_test.py

```
1  from flask import Flask
2  from flask import render_template
3  from flask import request
4
5  app = Flask(__name__)
6
7  @app.route("/hello")
8  def index():
9      name = request.args.get('name', 'Nikt')
10
11      if name:
12          greeting = f"Witaj, {name}"
13      else:
14          greeting = "Witaj świecie"
15
16      return render_template("index.html", greeting=greeting)
17
18  if __name__ == "__main__":
19      app.run()
```


Zrestartuj go (naciśnij *Ctrl*+*C*, a następnie uruchom ponownie), aby upewnić się, że ponownie się załaduje, a następnie w przeglądarce wpisz *http://localhost:5000/hello*. Powinien wyświetlić się komunikat: *Chciałem tylko powiedzieć Witaj, Nikt*. Następnie zmień adres URL w przeglądarce na *http://localhost:5000/hello?name=Franek*, a zobaczysz napis *Witaj, Franek*. Na koniec w części *name=Franek* adresu podstaw swoje imię. Teraz przeglądarka przywita się z Tobą.

Przeanalizujmy zmiany, które wprowadziłem do Twojego skryptu.

1. Zamiast zwykłego łańcucha znaków dla *greeting* używam teraz *request.args*, aby pobrać dane z przeglądarki. Jest to prosty słownik, który zawiera wartości formularza w postaci par klucz-wartość.
2. Następnie konstruuje *greeting* z nowego imienia *name*, co powinno być dla Ciebie teraz bardzo znajome.
3. Cała reszta pliku jest taka sama jak poprzednio.

Nie musisz też ograniczać się tylko do jednego parametru w adresie URL. Zmień ten przykład, aby podać dwie zmienne, tak jak w adresie *http://localhost:5000/hello?name=Franek&greet=Hola*. Następnie zmień kod, aby uzyskać *name* i *greet* w ten sposób:

```
greet = request.args.get('greet', 'Witaj')
greeting = f"{greet}, {name}"
```

Powinieneś również spróbować *nie podawać* parametrów *greet* i *name* w adresie URL. Po prostu wyślij przeglądarkę do *http://localhost:5000/hello*, aby zobaczyć, że *index* ma teraz domyślnie wartość „Nikt” dla *name* i „Witaj” dla *greet*.

Tworzenie formularzy HTML

Przekazywanie parametrów w adresie URL działa, ale jest to rozwiązanie brzydkie i niełatwe w użyciu dla zwykłych ludzi. Tym, czego naprawdę potrzebujesz, jest „formularz POST”, który jest specjalnym plikiem HTML zawierającym znacznik *<form>*. Ten formularz zbierze informacje od użytkownika, a następnie wyśle je do aplikacji internetowej tak samo, jak zrobiłeś powyżej.

Opracujmy jakiś szybki formularz, żebyś zobaczył, jak działa. Oto nowy plik HTML, który musisz utworzyć w *templates/hello_form.html*.

hello_form.html

```
<html>
  <head>
    <title>Przykładowy formularz internetowy</title>
  </head>
  <body>

    <h1>Wypełnij ten formularz</h1>
    <form action="/hello" method="POST">
      Powitanie: <input type="text" name="greet">
    <br/>
```

```

        Twoje imię: <input type="text" name="name">
    <br/>
    <input type="submit">
</form>

</body>
</html>

```

Następnie powinieneś zmienić plik *app.py*, aby wyglądał następująco.

app.py

```

1  from flask import Flask
2  from flask import render_template
3  from flask import request
4
5  app = Flask(__name__)
6
7  @app.route("/hello", methods=['POST', 'GET'])
8  def index():
9      greeting = "Witaj, świecie"
10
11     if request.method == "POST":
12         name = request.form['name']
13         greet = request.form['greet']
14         greeting = f"{greet}, {name}"
15         return render_template("index.html", greeting=greeting)
16     else:
17         return render_template("hello_form.html")
18
19
20 if __name__ == "__main__":
21     app.run()

```

Gdy już to zrobisz, po prostu uruchom ponownie aplikację internetową i wpisz jej adres w przeglądarce, tak jak przedtem.

Tym razem otrzymasz formularz z pytaniem o *Powitanie* i *Twoje imię*. Po naciśnięciu przycisku *Submit* w formularzu otrzymasz takie samo powitanie, jakie zwykle otrzymujesz, ale tym razem spójrz na adres URL w przeglądarce. Jest to *http://localhost:5000/hello*, mimo że wysłałeś parametry.

Działa to dzięki linii `<form action="/hello" method="POST">` z pliku *hello_form.html*. Jest to instrukcja dla przeglądarki, żeby wykonać następujące czynności.

1. Zebrać dane od użytkownika za pomocą pól formularza.
2. Wysłać je do serwera przy użyciu żądania POST, które jest po prostu kolejnym żądaniem przeglądarki „ukrywającym” pola formularza.
3. Wysłać to do adresu URL */hello* (tak jak pokazano w części `action="/hello"`).

Ponizej możesz zobaczyć, że dwa znaczniki `<input>` odpowiadają nazwom zmiennych w Twoim nowym kodzie. Zwróć również uwagę, że zamiast samej metody GET w `def index` mam również kolejną metodę, POST. Ta nowa aplikacja działa w następujący sposób.

1. Twoje żądanie trafia do `index()` jak zwykle, z tym wyjątkiem, że teraz istnieje instrukcja `if`, która sprawdza `request.method` dla metod POST lub GET. W ten sposób przeglądarka informuje *app.py*, że żądanie jest albo przesłaniem formularza, albo parametrami adresu URL.
2. Jeśli `request.method` to POST, przetwarzasz formularz tak, jakby został wypełniony i przesłany, zwracając właściwe powitanie.
3. Jeśli `request.method` jest czymkolwiek innym, zwracasz po prostu *hello_form.html* użytkownikowi do wypełnienia.

Jako ćwiczenie przejdź do pliku *templates/index.html* i dodaj link *powrotny* po prostu do */hello*, abyś mógł dalej wypełniać formularz i wyświetlać wyniki. Upewnij się, że potrafisz wyjaśnić, jak działa ten link i w jaki sposób pozwala Ci przełączać się między *templates/index.html* i *templates/hello_form.html* oraz co jest uruchamiane w tym najnowszym kodzie Pythona.

Tworzenie szablonu układu

Podczas pracy nad grą w następnym ćwiczeniu będziesz musiał utworzyć kilka niewielkich stron HTML. Pisanie pełnej strony internetowej za każdym razem szybko stanie się nudne. Na szczęście możesz utworzyć szablon „układu” lub inaczej rodzaj powłoki, która będzie opakowywać wszystkie inne strony wspólnymi nagłówkami i stopkami. Dobrzy programiści starają się zredukować powtarzalne czynności, więc układy są niezbędne, aby być dobrym programistą.

Zmień *templates/index.html*, aby wyglądał następująco.

index_laid_out.html

```
{% extends "layout.html" %}

{% block content %}

{% if greeting %}
    Chciałem tylko powiedzieć
    <em style="color: green; font-size: 2em;">{{ greeting }}</em>.
{% else %}
    <em>Witaj</em>, świecie!
{% endif %}

{% endblock %}
```

Następnie zmień *templates/hello_form.html*, aby wyglądał tak, jak poniżej.

hello_form_laid_out.html

```
{% extends "layout.html" %}

{% block content %}

<h1>Wypełnij ten formularz</h1>
```

```

<form action="/hello" method="POST">
    Powitanie: <input type="text" name="greet">
    <br/>
    Twoje imię: <input type="text" name="name">
    <br/>
    <input type="submit">
</form>

{% endblock %}

```

Ma to na celu pozbycie się „dodatkowego kodu” (ang. *boilerplate code*) na górze i na dole, który jest zawsze na każdej stronie. Wstawimy to z powrotem do pojedynczego pliku *templates/layout.html*, który od tej pory będzie obsługiwał to za nas.

Po wprowadzeniu tych zmian utwórz plik *templates/layout.html* z następującą zawartością.

layout.html

```

<html>
<head>
    <title>Goci z planety Percal 25</title>
</head>
<body>

{% block content %}

{% endblock %}

</body>
</html>

```

Ten plik wygląda jak zwykły szablon, z tym wyjątkiem, że przekazana będzie do niego zawartość pozostałych szablonów i będzie on używany do ich *opakowywania*. Cokolwiek tu umieścisz, nie będzie musiało znajdować się w innych szablonach. Twoje pozostałe szablony HTML będą wstawiane w sekcji `{% block content %}`. Framework flask używa pliku *layout.html* jako układu, ponieważ umieszczasz `{% extends "layout.html" %}` na początku swoich szablonów.

Pisanie zautomatyzowanych testów dla formularzy

Łatwo przetestować aplikację internetową za pomocą przeglądarki, wciskając po prostu przycisk odświeżania, ale w końcu jesteśmy programistami. Po co wykonywać jakieś powtarzające się zadania, kiedy możemy napisać kod, aby przetestować naszą aplikację? Teraz napiszesz mały test dla formularza aplikacji internetowej na podstawie tego, czego nauczyłeś się w ćwiczeniu 47. Jeśli nie pamiętasz tego ćwiczenia, przeczytaj je ponownie.

Utwórz nowy plik o nazwie *tests/app_tests.py* z następującą zawartością.

app_tests.py

```

1  from nose.tools import *
2  from app import app
3
4  app.config['TESTING'] = True
5  web = app.test_client()
6
7  def test_index():
8      rv = web.get('/', follow_redirects=True)
9      assert_equal(rv.status_code, 404)
10
11     rv = web.get('/hello', follow_redirects=True)
12     assert_equal(rv.status_code, 200)
13     assert_in(b"Wypełnij ten formularz", rv.data)
14
15     data = {'name': 'Zed', 'greet': 'Hola'}
16     rv = web.post('/hello', follow_redirects=True, data=data)
17     assert_in(b"Zed", rv.data)
18     assert_in(b"Hola", rv.data)

```

Na koniec użyj `nosetests`, aby uruchomić tę konfigurację testową i przetestować aplikację internetową.

```
$ nosetests
```

```
.
```

```
-----
Ran 1 test in 0.059s
```

```
OK
```

To, co tutaj robię, polega w rzeczywistości na *zaimportowaniu* całej aplikacji z modułu `app.py`, a następnie uruchomieniu jej ręcznie. Framework `flask` ma bardzo prosty interfejs API do przetwarzania żądań, który wygląda tak:

```

data = {'name': 'Zed', 'greet': 'Hola'}
rv = web.post('/hello', follow_redirects=True, data=data)

```

Oznacza to, że możesz wysłać żądanie POST za pomocą metody `post()`, a następnie przekazać dane formularza jako słownik. Wszystko inne działa tak samo jak testowanie żądań `web.get()`.

W zautomatyzowanym teście `tests/app_tests.py` najpierw upewniam się, że adres / URL zwraca odpowiedź `404 Not Found`, ponieważ w rzeczywistości nie istnieje. Następnie sprawdzam, czy `/hello` działa zarówno z GET, jak i z formularzem POST. Prześledzenie testu powinno być dość proste nawet wtedy, kiedy całkowicie nie będziesz miał pojęcia, co się dzieje w nim dzieje.

Poświęć trochę czasu na zapoznanie się z tą najnowszą aplikacją, zwłaszcza ze sposobem działania zautomatyzowanych testów. Upewnij się, że rozumiesz, jak zaimportowałem aplikację z `app.py` i uruchomiłem ją bezpośrednio dla zautomatyzowanego testu. Jest to ważna sztuczka, która stanowi podstawę do dalszej nauki.

Zrób to sam

1. Poczytaj jeszcze więcej o HTML-u i nadaj temu prostemu formularzowi jeszcze lepszy układ. Łatwiej Ci będzie, jeśli najpierw rozrysujesz na papierze, czego potrzebujesz, a *następnie* zaimplementujesz to za pomocą kodu HTML.
2. To będzie trudne, ale spróbuj dowiedzieć się, jak zrobić formularz przesyłania plików, aby można było przesłać obraz i zapisać go na dysku.
3. To będzie jeszcze bardziej odmóżdżające, ale znajdź HTTP RFC (czyli dokument, który opisuje sposób działania HTTP) i przeczytaj tyle, ile dasz radę. To jest naprawdę nudne, ale przydaje się raz na jakiś czas.
4. To również będzie bardzo trudne, ale sprawdź, czy możesz znaleźć kogoś, kto pomoże Ci skonfigurować serwer WWW, taki jak Apache, Nginx lub thttpd. Spróbuj zaserwować za jego pomocą kilka plików *.html* i *.css*, aby zobaczyć, czy Ci się uda. Nie przejmuj się, jeśli nie dasz rady. Serwery WWW w pewnym sensie są do bani.
5. Zrób sobie przerwę, a potem spróbuj zrobić tyle różnych aplikacji internetowych, ile zdołasz.

Popsuj kod

To świetny czas, aby dowiedzieć się, jak psuć aplikacje internetowe. Powinieneś poeksperymentować z następującymi rzeczami.

1. Jak dużo szkód możesz wyrządzić przy włączonym ustawieniu `FLASK_DEBUG`? Uważaj, abyś przy tej okazji sam się „nie wyczyścił”.
2. Załóżmy, że nie masz domyślnych parametrów dla formularzy. Co mogło pójść źle?
3. Sprawdzasz `POST`, a następnie „wszystko inne”. Możesz użyć narzędzia wiersza poleceń `curl` do wygenerowania różnych typów żądań. Co się dzieje?

Początek Twojej gry internetowej

Zbliżamy się do końca książki i w tym ćwiczeniu zamierzam naprawdę rzucić Ci wyzwanie. Kiedy skończysz, będziesz w miarę kompetentnym, początkującym programistą Pythona. Oczywiście będziesz musiał przeczytać jeszcze kilka innych książek i napisać kilka kolejnych projektów, ale będziesz posiadał umiejętności, aby je ukończyć. Jedynymi przeszkodami będą czas, motywacja i zasoby.

W tym ćwiczeniu nie zrobimy kompletnej gry, ale opracujemy „silnik”, który może uruchamiać grę z ćwiczenia 43. w przeglądarce. Ćwiczenie będzie obejmować refaktoryzację ćwiczenia 43., domieszanie struktury z ćwiczenia 47., dodanie zautomatyzowanych testów, a na koniec utworzenie silnika internetowego, który może uruchamiać gry.

To ćwiczenie będzie *olbrzymie* i przewiduję, że możesz poświęcić na nie równie dobrze tydzień albo i parę miesięcy, zanim przejdiesz dalej. Najlepiej podzielić je na małe kawałki i robić po trochu każdego wieczoru, nie spiesząc się, aby wszystko prawidłowo zadziało.

Refaktoryzacja gry z ćwiczenia 43.

Zmieniałeś projekt *gothonweb* dla dwóch ćwiczeń, a teraz w tym ćwiczeniu zrobisz to jeszcze raz. Umiejętność, której właśnie się uczysz, nazywa się „refaktoryzacją” lub — jak ja to lubię nazywać — „naprawianiem różnych rzeczy”. Refaktoryzacja to termin, którego programiści używają do opisu procesu pracy nad starym kodem i zmieniania go, aby dodać nowe funkcje lub po prostu wyczyścić. Robiłeś już to, nawet o tym nie wiedząc, ponieważ jest to druga natura budowania oprogramowania.

Z ćwiczenia 47. zacerpniemy pomysły testowalnej „mapy” pokoi, a z ćwiczenia 43. weźmiemy grę i połączymy te rzeczy ze sobą, aby utworzyć nową strukturę gry. Będzie ona miała tę samą treść, tyle że „zrefaktoryzowaną”, aby uzyskać lepszą strukturę.

Pierwszym krokiem jest pobranie kodu z *ex47/game.py* i skopiowanie go do *gothonweb/planisphere.py*, skopiowanie pliku *tests/ex47_tests.py* do *tests/planisphere_tests.py* oraz ponowne uruchomienie *nosetests*, aby upewnić się, że wszystko działa. Angielskie słowo *planisphere* (planisfera) jest synonimem słowa *map* (mapa) i pozwala uniknąć kolizji nazw z wbudowaną funkcją *map* Pythona. Tezaurus jest Twoim przyjacielem.

OSTRZEŻENIE! Nie będę już pokazywał danych wyjściowych z uruchamiania testów. Po prostu załóż, że powinienes to zrobić i będzie wyglądało tak, jak poprzednio, chyba że otrzymasz błąd.

Gdy masz już skopiowany kod z ćwiczenia 47., nadszedł czas na jego zrefaktoryzowanie, aby umieścić w nim mapę z ćwiczenia 43. Zacznę od określenia podstawowej struktury, a potem Twoim zadaniem będzie dokończenie plików *planisphere.py* i *planisphere_tests.py*.

Ułóż podstawową strukturę mapy, używając klasy Room w jej obecnej postaci.

planisphere.py

```
1  class Room(object):
2
3      def __init__(self, name, description):
4          self.name = name
5          self.description = description
6          self.paths = {}
7
8      def go(self, direction):
9          return self.paths.get(direction, None)
10
11     def add_paths(self, paths):
12         self.paths.update(paths)
13
14
15     central_corridor = Room("Centralny korytarz",
16     """
17     Goci z Planety Percal 25 wdarli się na Twój statek i wymordowali całą
18     załogę. Jesteś ostatnim ocalałym członkiem załogi, a Twoją ostatnią misją
19     jest zdobycie bomby neutronowej z magazynu broni, umieszczenie jej na mostku
20     i wysadzenie statku, gdy już uda Ci się dotrzeć do kapsuły ratunkowej.
21
22     Biegniesz centralnym korytarzem w kierunku magazynu broni, gdy nagle
23     wyskakuje jakiś Got o czerwonej, łuszczącej się skórze, z czarnymi zębami,
24     ubrany w kostium złego klauna i cały ziejący nienawiścią. Blokuje dostęp
25     do magazynu broni i właśnie zamierza wyciągnąć broń, żeby Cię rozwalić.
26     """)
27
28
29     laser_weapon_armory = Room("Magazyn broni laserowej",
30     """
31     Na Twoje szczęście w akademii nauczyli Cię rzucać mięsem w obcych językach.
32     Opowiadasz jedyny gocki żart, jaki znasz: Lbhe zbgure vf fb sng, jura fur
33     fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr. Got zastyga, przez chwilę
34     próbuje się powstrzymać, a następnie wybucha śmiechem i nie może się ruszyć.
35     W tym czasie szybko uciekasz i na odchodnym strzelasz mu prosto w głowę,
36     powalając go trupem, a następnie znikasz za drzwiami magazynu broni.
37
38     Dajesz nura do magazynu broni, przykucasz i lustrujesz wzrokiem pomieszczenie,
39     szukając ukrywających się Gotów. Panuje martwa, przerażająca cisza. Wstajesz
40     i biegniesz w odległy koniec pomieszczenia, gdzie znajdujesz bombę neutronową
41     umieszczoną w zabezpieczonym pojemniku. Aby wyjąć bombę, musisz odblokować
42     zamek, wpisując na klawiaturze kod. Jeśli 10 razy wpiszesz niewłaściwy kod,
43     zamek zablokuje się na zawsze i nie wyjmiesz bomby. Kod ma 3 cyfry.
44     """)
45
46
47     the_bridge = Room("Mostek",
48     """
49     Blokada puszcza i pojemnik otwiera się, z sykiem wypuszczając ze środka gaz.
```

```
50 Chwytasz bombę neutronową i biegniesz tak szybko, jak możesz, w kierunku
51 mostka, gdzie musisz podłożyć ją w odpowiednim miejscu.
52
53 Zdyszany wpadasz na mostek z bombą neutronową pod pachą i zaskakujesz
54 5 Gotów, którzy próbują przejąć kontrolę nad statkiem. Każdy kolejny
55 ma jeszcze brzydszy kostium klauna niż poprzedni. Żaden z nich nie
56 wyciągnął jeszcze broni, ponieważ widzą aktywowaną bombę pod Twoją
57 pachą i nie chcą jej przypadkowo zdetonować.
58 """)
59
60
61 escape_pod = Room("Kapsuła ratunkowa",
62 """)
63 Przykładasz lufę miotacza do trzymanej pod pachą bomby, a Goci podnoszą
64 ręce do góry i zaczynają się pocić. Cofasz się powoli do drzwi, otwierasz
65 je, a potem ostrożnie kładziesz bombę na podłodze, nadal celując w nią
66 miotaczem. Następnie przeskakujesz przez drzwi, zamykasz je przyciskiem i
67 strzelasz w panel kontrolny zamka, aby nie mogli się wydostać. Bomba
68 podłożona, więc uciekasz do kapsuły ratunkowej, aby wydostać się z tej puszk.
69
70 Biegniesz korytarzami, rozpaczliwie próbując dotrzeć do kapsuły, zanim
71 cały statek eksploduje. Wydaje się, że na statku nie ma już prawie
72 żadnego Gota, więc masz wolną drogę ucieczki. Docierasz do komory z
73 kapsułami ratunkowymi i teraz musisz zdecydować się na jedną z nich.
74 Niektóre mogły zostać uszkodzone, ale nie masz czasu sprawdzać. Jest 5
75 kapsuł, którą wybierasz?
76 """)
77
78
79 the_end_winner = Room("Koniec",
80 """)
81 Wskakujesz do kapsuły nr 2 i katapultujesz się. Kapsuła lekko wślizguje się w
82 kosmos, zmierzając w kierunku poniższej planety. W trakcie lotu spoglądasz wstecz
83 i widzisz, że Twój statek imploduje, a następnie eksploduje, jak jasna gwiazda,
84 niszcząc jednocześnie statek Gotów. Wygrałeś!
85 """)
86
87
88 the_end_loser = Room("Koniec",
89 """)
90 Wskakujesz do losowej kapsuły i katapultujesz się. Kapsuła wystrzeliwuje
91 w otchłań kosmosu, a następnie imploduje, gdy kadłub pęka i miażdży Twoje
92 ciało, robiąc z niego dżem malinowy.
93 """)
94 )
95
96 escape_pod.add_paths({
97     '2': the_end_winner,
98     '*': the_end_loser
99 })
100
101 generic_death = Room("death", "Umarłeś.")
102
103 the_bridge.add_paths({
```

```

104         'rzucam bombę': generic_death,
105         'powoli podkładam bombę': escape_pod
106     })
107
108     laser_weapon_armory.add_paths({
109         '0132': the_bridge,
110         '*': generic_death
111     })
112
113     central_corridor.add_paths({
114         'strzelam!': generic_death,
115         'robię unik!': generic_death,
116         'opowiadam dowcip': laser_weapon_armory
117     })
118
119     START = 'central_corridor'
120
121     def load_room(name):
122         """
123         W tym miejscu istnieje potencjalny problem z zabezpieczeniami.
124         Kto ma ustawić name? Czy to może narazić zmienną?
125         """
126         return globals().get(name)
127
128     def name_room(room):
129         """
130         Ten sam potencjalny problem z zabezpieczeniami. Czy można zaufać room?
131         Jakie jest lepsze rozwiązanie niż to sprawdzanie zmiennych globalnych?
132         """
133         for key, value in globals().items():
134             if value == room:
135                 return key

```

Zauważasz teraz, że jest kilka problemów z naszą klasą Room i tą mapą.

1. Tekst, który był umieszczony w klauzulach if-else i drukowany *przed* wejściem do pokoju, musimy wstawić jako część każdego pokoju. Oznacza to, że nie możesz zmieniać planisfery, co byłoby miłe. Naprawisz to w tym ćwiczeniu.
2. W oryginalnej grze znajdują się fragmenty, gdzie uruchamialiśmy kod, który określał takie rzeczy, jak kod do klawiatury bomby lub właściwą kapsułę. W tej grze wybieramy po prostu pewne wartości domyślne i ich używamy, ale później w „Zrób to sam” będziesz musiał sprawić, żeby to znów działało.
3. Właśnie utworzyłem zakończenie generic_death (ogólna śmierć) dla wszystkich złych decyzji i musisz je za mnie dokończyć. Musisz wrócić, dodać wszystkie oryginalne zakończenia i upewnić się, że działają.
4. Mam tutaj nowy rodzaj przechodzenia przez stany gry oznaczony "*", który będzie używany do „uniwersalnej” akcji w silniku.

Kiedy już to w zasadzie napiszesz, masz tu nowy zautomatyzowany test, *tests/planisphere_test.py*, który powinienś dodać, aby zacząć.

```
1  from nose.tools import *
2  from gothonweb.planisphere import *
3
4  def test_room():
5      gold = Room("GoldRoom",
6                  """W tym pokoju jest złoto, które możesz zabrać. Na
7                  północy są drzwi.""")
8      assert_equal(gold.name, "GoldRoom")
9      assert_equal(gold.paths, {})
10
11  def test_room_paths():
12      center = Room("Center", "Test pokoju pośrodku.")
13      north = Room("North", "Test pokoju na północy.")
14      south = Room("South", "Test pokoju na południu.")
15
16      center.add_paths({'north': north, 'south': south})
17      assert_equal(center.go('north'), north)
18      assert_equal(center.go('south'), south)
19
20  def test_map():
21      start = Room("Start", "Możesz iść na zachód i w dół.")
22      west = Room("Trees", "Tutaj są trzy drzewa, możesz iść na wschód.")
23      down = Room("Dungeon", "Tu na dole jest ciemno, możesz iść do góry.")
24
25      start.add_paths({'west': west, 'down': down})
26      west.add_paths({'east': start})
27      down.add_paths({'up': start})
28
29      assert_equal(start.go('west'), west)
30      assert_equal(start.go('west').go('east'), start)
31      assert_equal(start.go('down').go('up'), start)
32
33  def test_gothon_game_map():
34      start_room = load_room(START)
35      assert_equal(start_room.go('strzelam!'), generic_death)
36      assert_equal(start_room.go('robię unik!'), generic_death)
37
38      room = start_room.go('opowiadam dowcip')
39      assert_equal(room, laser_weapon_armory)
```

Twoim zadaniem w tej części ćwiczenia jest ukończenie mapy i postaranie się, aby zautomatyzowany test kompletnie weryfikował całą mapę. Obejmuje to poprawienie wszystkich obiektów `generic_death`, żeby były prawdziwymi zakończeniami. Upewnij się, że działa to naprawdę dobrze i że Twój test jest możliwie kompletny, ponieważ będziemy zmieniać tę mapę później i użyjesz tych testów, aby upewnić się, że wszystko wciąż działa.

Tworzenie silnika

Powinieneś już mieć działającą mapę gry i napisany dla niej dobry test jednostkowy. Teraz chcę, żebyś przygotował prosty, mały silnik gry, który będzie uruchamiał pokoje, zbierał dane wejściowe od gracza i śledził, w którym miejscu gry znajduje się gracz. Wykorzystamy sesję, której się właśnie nauczyłeś, aby utworzyć prosty silnik gry wykonujący następujące czynności.

1. Rozpoczynanie nowej gry dla nowych użytkowników.
2. Prezentowanie pokoju użytkownikowi.
3. Przyjmowanie danych wejściowych od użytkownika.
4. Uruchamianie danych wprowadzanych przez użytkownika przez całą grę.
5. Wyświetlanie wyników i kontynuowanie działania dopóki, dopóty postać gracza nie umrze.

Aby to zrobić, weźmiesz zaufaną aplikację *app.py*, którą hakowałeś, i utworzysz w pełni działający, oparty na sesjach silnik gry. Haczyk polega na tym, że ja zrobię bardzo prosty silnik z *podstawowymi plikami HTML*, a do Ciebie będzie należało jego dokończenie. Oto podstawowy silnik.

app.py

```

1  from flask import Flask, session, redirect, url_for, escape, request
2  from flask import render_template
3  from gothonweb import planisphere
4
5  app = Flask(__name__)
6
7  @app.route("/")
8  def index():
9      # to jest używane do "skonfigurowania" sesji z wartościami początkowymi
10     session['room_name'] = planisphere.START
11     return redirect(url_for("game"))
12
13  @app.route("/game", methods=['GET', 'POST'])
14  def game():
15     room_name = session.get('room_name')
16
17     if request.method == "GET":
18         if room_name:
19             room = planisphere.load_room(room_name)
20             return render_template("show_room.html", room=room)
21         else:
22             # dlaczego to tutaj jest? potrzebujesz tego?
23             return render_template("you_died.html")
24     else:
25         action = request.form.get('action')
26
27         if room_name and action:
28             room = planisphere.load_room(room_name)
29             next_room = room.go(action)
30
31         if not next_room:
```

```

32         session['room_name'] = planisphere.name_room(room)
33     else:
34         session['room_name'] = planisphere.name_room(next_room)
35
36     return redirect(url_for("game"))
37
38
39 # POWINIENES TO ZMIENIĆ, JEŚLI UMIEŚCISZ GRĘ W INTERNECIE
40 app.secret_key = 'A0Zr98j/3yX R-XHH!jmN]LWX/,?RT'
41
42 if __name__ == "__main__":
43     app.run()

```

W tym skrypcie jest jeszcze więcej nowych rzeczy, ale — o dziwo — jest to cały internetowy silnik gry w jednym niewielkim pliku. Zanim uruchomisz aplikację *app.py*, musisz zmienić swoją zmienną środowiskową PYTHONPATH. Nie wiesz, co to jest? Wiem, to trochę głupie, ale musisz nauczyć się, co to jest, aby uruchamiać nawet podstawowe programy Pythona — tak lubią robić pythonowcy.

W systemach Linux lub macOS w terminalu wpisz:

```
export PYTHONPATH=$PYTHONPATH:.
```

W systemach Windows w programie PowerShell wpisz:

```
$env:PYTHONPATH = "$env:PYTHONPATH;."
```

Wystarczy to zrobić tylko raz na sesję powłoki, ale jeśli pojawi się błąd importu, prawdopodobnie musisz to zrobić jeszcze raz lub zrobiłeś to źle.

Następnie powinieneś usunąć *templates/hello_form.html* oraz *templates/index.html* i utworzyć dwa szablony wymienione w poprzednim kodzie. Oto bardzo prosty szablon *templates/show_room.html*.

show_room.html

```

{% extends "layout.html" %}

{% block content %}

<h1> {{ room.name }} </h1>

<pre>
{{ room.description }}
</pre>

{% if room.name in ["death", "Koniec"] %}
    <p><a href="/">Grasz jeszcze raz?</a></p>
{% else %}
    <p>
        <form action="/game" method="POST">
            - <input type="text" name="action"> <input type="SUBMIT">
        </form>
    </p>
{% endif %}

{% endblock %}

```

Ten szablon służy do pokazywania pokoju, gdy użytkownik podróżuje przez grę. Następnie potrzebujesz szablonu do informowania gracza, że umarł, w przypadku gdy przypadkowo dojdzie do końca mapy. Będzie to *templates/you_died.html*.

you_died.html

```
<h1>Umarłeś!</h1>
```

```
<p>Wygląda na to, że poszedłeś do piachu.</p>
```

```
<p><a href="/">Grasz jeszcze raz?</a></p>
```

Gdy już masz to gotowe, powinieneś być w stanie wykonać następujące czynności.

1. Spraw, aby *tests/app_tests.py* znowu zadziałał, żeby przetestować grę. Nie będziesz w stanie zrobić w grze więcej niż kilka kliknięć z powodu sesji, ale powinieneś zrobić podstawowe rzeczy.
2. Uruchom skrypt `python3.6 app.py` i przetestuj grę.

Powinieneś mieć możliwość odświeżania i naprawiania gry, jak zwykle. Powinno także dać się pracować z kodem HTML i silnikiem gry, dopóki sama gra nie będzie robiła wszystkich wymaganych rzeczy.

Twój egzamin końcowy

Czy czujesz, że nagle otrzymałeś ogromną ilość informacji? To dobrze, chcę, żebyś miał nad czym majsterkować, gdy będziesz budował swoje umiejętności. Aby ukończyć to ćwiczenie, dam Ci ostateczny zestaw zadań, które musisz wykonać samodzielnie. Zauważysz, że to, co napisałeś do tej pory, nie jest zbyt dobrze zbudowane. To tylko pierwsza wersja kodu. Twoim zadaniem jest teraz sprawić, aby gra stała się bardziej kompletna i w tym celu zrób następujące rzeczy.

1. Napraw w kodzie wszystkie błędy, o których wspominałem, i te, o których nie wspominałem. Jeśli znajdziesz nowe błędy, daj mi znać.
2. Ulepsz wszystkie zautomatyzowane testy, aby testować większą część aplikacji. Postaraj się dojść do punktu, w którym do sprawdzania aplikacji podczas pracy będziesz używał testu, a nie przeglądarki.
3. Spraw, aby kod HTML wyglądał lepiej.
4. Poszukaj informacji na temat logowania i utwórz system logowania dla aplikacji, aby gracze mogli mieć loginy i zapisywać wyniki.
5. Dokończ mapę gry, aby była jak największa i posiadała możliwie dużo funkcjonalności.
6. Zapewnij użytkownikom system „pomocy”, który pozwala im zapytać, co mogą zrobić w każdym pokoju w grze.
7. Dodaj do gry wszystkie inne funkcjonalności, jakie przyjdą Ci do głowy.
8. Utwórz kilka „map” i pozwól użytkownikom wybierać grę, którą chcą uruchomić. Twój silnik *app.py* powinien być w stanie uruchomić dowolną mapę pokoi, którą mu podasz, żebyś mógł obsługiwać wiele gier.

9. Na koniec użyj tego, czego nauczyłeś się w ćwiczeniach 48. i 49., aby utworzyć lepszy procesor danych wejściowych. Masz już większość niezbędnego kodu. Musisz tylko poprawić gramatykę i połączyć to z formularzem danych wejściowych i silnikiem gry.

Powodzenia!

Typowe pytania

Używam sessions w mojej grze i nie mogę tego przetestować za pomocą nosetests.

Informacje na temat tworzenia fałszywych sesji w testach znajdziesz w dokumentacji *Testing Flask Applications*, w sekcji *Other Testing Tricks* (<http://flask.pocoo.org/docs/0.12/testing/#other-testing-tricks>).

Otrzymuję błąd ImportError. Może to być spowodowane jedną z następujących rzeczy (lub kilkoma naraz): złym katalogiem, złą wersją Pythona, brakiem ustawienia zmiennej środowiskowej PYTHONPATH, brakiem pliku `__init__.py` lub błędami w piśmowni w importach.

Następne kroki

Nie jesteś jeszcze programistą. Lubię myśleć o tej książce jak o przyznaniu Ci „czarnego pasa w programowaniu”. Wiesz już wystarczająco dużo, aby rozpocząć lekturę kolejnej książki o programowaniu i dobrze sobie poradzić. Ta książka powinna dać Ci mentalne narzędzia i podejście, którego potrzebujesz, aby przebrnąć przez większość książek o Pythonie i rzeczywiście się czegoś nauczyć. Może Ci nawet wiele ułatwić.

Polecam zapoznanie się z niektórymi z poniższych projektów i podjęcie próby zbudowania czegoś z ich pomocą.

- *Learn Ruby The Hard Way* (<https://learnrubythehardway.org>): nauczysz się jeszcze więcej o programowaniu, gdy będziesz poznawał kolejne języki programowania, więc spróbuj nauczyć się również języka Ruby.
- Samouczek Django (<https://docs.djangoproject.com/en/1.11/intro/>): zbuduj aplikację internetową za pomocą frameworku webowego Django.
- SciPy (<https://www.scipy.org>): zapoznaj się z tym, jeśli interesujesz się nauką, matematyką i inżynierią.
- PyGame (<http://www.pygame.org>): utwórz grę z grafiką i dźwiękiem.
- Pandas (<http://pandas.pydata.org>): użyj tego do manipulowania danymi i przeprowadzania analizy.
- Natural Language Toolkit (<http://www.nltk.org>): użyj tego do analizy tekstu pisanego i pisanego na przykład filtrów spamu i chat botów.
- TensorFlow (<https://www.tensorflow.org>): użyj tego do uczenia maszynowego i wizualizacji.
- REquests (<http://docs.python-requests.org>): poznaj HTTP i sieć WWW po stronie klienta.
- ScraPy (<https://scrapy.org>): spróbuj scrapingu stron internetowych, aby pobierać z nich informacje.
- Kivy (<https://kivy.org>): twórz interfejsy użytkownika na komputerach stacjonarnych i platformach mobilnych.
- *Programowanie w C. Sprytnie podejście do trudnych zagadnień, których wolałbyś unikać (takich jak język C)* (<https://helion.pl/ksiazki/pcspry>): po zapoznaniu się z Pythonem spróbuj nauczyć się języka C i algorytmów z mojej kolejnej książki. Nie śpiesz się. Język C jest inny, ale warto go poznać.

Wybierz jeden z powyższych zasobów i przejrzyj wszystkie zamieszczone tam poradniki oraz całą dokumentację. Gdy czytasz dokumentację zawierającą kod, *wpisuj cały kod* i uruchamiaj go. Ja właśnie tak robię. Tak robi każdy programista. Czytanie dokumentacji programistycznej nie wystarczy, aby się jej nauczyć. Musisz to przećwiczyć. Po zapoznaniu się z samouczkiem i wszelką dostępną dokumentacją spróbuj coś zbudować. Może to być cokolwiek, nawet coś, co ktoś już napisał. Po prostu coś zrób.

Musisz zrozumieć, że cokolwiek napiszesz, prawdopodobnie będzie do niczego. Jednak nie przejmuj się tym. Na początku nie byłem doskonały i nie znałem każdego języka programowania. Żaden początkujący nie pisze od razu czystego, wysokiej próby złota i każdy, kto twierdzi, że jemu się to udało, jest wielkim łgarzem.

Jak uczyć się dowolnego języka programowania

Pokażę Ci, jak nauczyć się większości języków programowania, które może zechcesz poznać w przyszłości. Organizacja tej książki opiera się na sposobie, w jakim ja i wielu innych programistów uczymy się nowych języków. Zazwyczaj przechodzę przez następujący proces.

1. Zdobądź książkę lub jakieś wstępne opracowanie na temat języka.
2. Przestudiuj tę książkę i wpisz cały kod, upewniając się, że wszystko działa.
3. Czytaj książkę podczas pracy nad kodem i rób notatki.
4. Użyj danego języka do zaimplementowania niewielkiego zestawu programów, które znasz, ale napisane zostały w innym języku.
5. Czytaj cudzy kod w tym języku i spróbuj kopiować z niego wzorce kodowania.

W tej książce zmusiłem Cię do przejścia przez ten proces bardzo powoli i małymi porcjami. Inne książki nie są tak zorganizowane, a to oznacza, że musisz ekstrapolować to podejście na ich treść. Najlepszym sposobem jest luźne przeczytanie książki i sporządzenie listy wszystkich głównych sekcji kodu. Przekształć tę listę w zestaw ćwiczeń na podstawie rozdziałów, a następnie po prostu wykonaj po kolei, po jednym na raz.

Opisany powyżej proces sprawdza się również w przypadku nowych technologii, pod warunkiem że istnieją na ich temat jakieś książki, które możesz przeczytać. Jeśli nie ma żadnych książek dotyczących interesującego Cię tematu, wykonuj ten proces, ale jako wprowadzenia używaj dokumentacji z internetu lub kodu źródłowego.

Każdy nowo poznany język sprawia, że jesteś lepszym programistą, a nauka kolejnych staje się łatwiejsza. Przy trzecim lub czwartym języku powinieneś poznawać zbliżone języki w tydzień, przy czym więcej czasu będziesz musiał poświęcić na całkowicie obce języki. Skoro znasz już Pythona, możesz potencjalnie dość szybko nauczyć się przez porównanie języków Ruby i JavaScript. Jest tak dlatego, że wiele języków ma wspólne, zbliżone koncepcje, a kiedy poznasz je w jednym języku, będą działać też w innych.

Ostatnią rzeczą do zapamiętania na temat nauki nowego języka jest zasada: „Nie bądź głupim turystą”. Głupi turysta to ktoś taki, kto jedzie do innego kraju, a potem narzeka, że jedzenie nie jest takie jak w domu („Dlaczego w tym głupim kraju nie mogę dostać dobrego burgera?!”). Kiedy uczysz się nowego języka, zakładaj, że to, co on robi, nie jest głupie, jest po prostu inne, więc zaakceptuj to i ucz się.

Gdy jednak nauczysz się jakiegoś języka, nie bądź niewolnikiem jego sposobu działania. Czasami ludzie, którzy używają określonego języka, robią naprawdę idiotyczne rzeczy tylko i wyłącznie dlatego, że „zawsze robiliśmy to w ten sposób”. Jeśli podoba Ci się Twój styl i wiesz, jak wszyscy inni to robią, możesz łamać ich zasady, kiedy potrafisz coś poprawić.

Bardzo lubię uczyć się nowych języków programowania. Uważam się za „programistę antropologa” i traktuję języki jak niewielkie zasoby wiedzy na temat grup programistów, którzy z nich korzystają. Uczę się języka, którego wszyscy używają, aby rozmawiać przy użyciu komputerów i uważam to za fascynujące. Jak jednak wiadomo, jestem dość dziwnym facetem, więc po prostu uczę się języków programowania, ponieważ tego chcesz.

Miłej zabawy! To naprawdę fajne rzeczy.

Porada starego programisty

Skończyłeś tę książkę i zdecydowałeś się kontynuować naukę programowania. Może to będzie dla Ciebie stopień w karierze zawodowej, a może hobby. Będziesz potrzebował porady, aby upewnić się, że podążasz właściwą ścieżką i czerpiesz jak największą przyjemność z nowo wybranej działalności.

Programuję od bardzo dawna. Tak długo, że jest to dla mnie niesamowicie nudne. Kiedy pisałem tę książkę, znałem około 20 języków programowania i mogłem nauczyć się dowolnego nowego języka w jeden dzień lub tydzień w zależności od tego, jak dziwne były to języki. W końcu jednak stało się to nudne i nie potrafiłem już obudzić w sobie zainteresowania. To nie znaczy, że myślę, iż programowanie *jest* nudne lub *Ty* pomyślisz, że jest nudne. Znaczy to tylko tyle, że *ja* uważam to za nieinteresujące w tym momencie mojej podróży.

Po odbyciu podróży, w której uczyłem się języków, odkryłem, że to nie języki mają znaczenie, ale to, co z nimi robisz. Właściwie zawsze to wiedziałem, ale języki mnie rozpraszały i na pewien czas o tym zapomniałem. Teraz nigdy tego nie zapomnę i *Ty* też nie powinienes.

Którego języka programowania się uczysz i używasz, nie ma znaczenia. *Nie* daj się wciągnąć w religie otaczające języki programowania, które tylko przesłaniają Ci ich prawdziwy cel. Jest nim bycie Twoim narzędziem do robienia ciekawych rzeczy.

Programowanie jako działalność intelektualna jest *jedyną* formą sztuki, która pozwala tworzyć interaktywną sztukę. Możesz opracowywać projekty, którymi będą bawić się inni ludzie i możesz z nimi rozmawiać w sposób pośredni. Żadna inna forma sztuki nie jest tak interaktywna. Filmy trafiają do odbiorców w jednym kierunku. Obrazy nie ruszają się. Kod płynie w obie strony.

Programowanie jako zawód jest tylko umiarkowanie interesujące. To może być dobra robota, ale możesz zarobić podobne pieniądze i być szczęśliwszy, prowadząc jakiś fast food. Lepiej używać kodu jako tajnej broni w innym zawodzie.

W świecie firm technologicznych ludzi, którzy potrafią kodować, można liczyć na pięćki i nie cieszą się szacunkiem. Jednak w biologii, medycynie, administracji, socjologii, fizyce, historii czy matematyce ludzie, którzy potrafią kodować, są szanowani i potrafią robić niesamowite rzeczy, aby rozwijać te dyscypliny.

Oczywiście wszystkie te porady są bezcelowe. Jeśli podoba Ci się nauka pisania oprogramowania z tą książką, powinienes użyć tego, aby poprawić swoje życie w każdy możliwy sposób. Idź i badaj tę dziwną, cudowną, nową, intelektualną pasję, której prawie nikt w ciągu ostatnich 50 lat nie był w stanie zgłębić. Ciesz się tym, póki możesz.

Na koniec powiem, że nauka tworzenia oprogramowania zmienia Cię i sprawia, że stajesz się kimś innym. Nie lepszym lub gorszym, po prostu innym. Może się okazać, że ludzie potraktują Cię szorstko, ponieważ potrafisz tworzyć oprogramowanie i być może będą używać wobec Ciebie takich słów jak „maniak komputerowy”. Być może przekonasz się, że skoro jesteś

w stanie rozłożyć ich logikę na czynniki pierwsze, nie będą chcieli z Tobą dyskutować. Może się nawet okazać, że samo poznanie sposobu działania komputera sprawi, iż staniesz się dla nich irytujący i dziwny.

Na to mam tylko jedną radę: „Niech idą do diabła!”. Świat potrzebuje więcej dziwnych ludzi, którzy wiedzą, jak działają różne rzeczy, i lubią się nad tym wszystkim zastanawiać. Kiedy będą Cię tak traktować, po prostu pamiętaj, że to jest *Twoja* podróż, nie ich. Bycie innym nie jest przestępstwem, a ludzie, którzy mówią Ci, że tak jest, są po prostu zazdrośni, iż zdobyłeś umiejętności, o których oni nie mogli śnić nawet w najśmielszych snach.

Możesz kodować. Oni nie mogą. To cholernie fajne.

Przyspieszony kurs wiersza poleceń

Dodatek ten jest superszybkim kursem korzystania z wiersza poleceń. Należy go wykonać dość szybko, w dzień lub dwa; nie jest to zaawansowany kurs używania powłoki.

Wprowadzenie: zamknij się i zacznij używać powłoki

Dodatek to przyspieszony kurs używania wiersza polecenia, aby zmusić komputer do wykonywania określonych zadań. Jako kurs przyspieszony nie jest tak szczegółowy ani obszerny jak moje pozostałe książki. Został zaprojektowany w taki sposób, abyś mógł zaledwie zacząć korzystać z komputera tak, jak robi to prawdziwy programista. Kiedy skończysz pracę z tym dodatkiem, będziesz potrafił wydawać większość podstawowych poleceń, z których każdy użytkownik powłoki korzysta na co dzień. Poznasz podstawy obsługi katalogów i kilka innych koncepcji.

Jedyna rada, którą Ci dam, jest następująca:

Zamknij się i wpiszuj wszystkie ćwiczenia.

Przepraszam, że jestem okrutny, ale to właśnie musisz zrobić. Jeśli masz irracjonalny lęk przed wierszem poleceń, to jedynym sposobem na jego pokonanie jest po prostu zacisnąć zęby i walczyć.

Nie zniszczysz komputera. Nie zostaniesz wrzucony do żadnego więzienia w podziemiach campusu Redmond Microsoftu. Twoi przyjaciele nie będą się z Ciebie śmiać, że jesteś frajerem. Po prostu zignoruj wszystkie głupie, dziwne powody, dla których obawiasz się wiersza poleceń.

Czemu? Powód jest jeden. Jeśli chcesz nauczyć się kodować, musisz opanować pracę z wierszem poleceń. Języki programowania to zaawansowane sposoby sterowania komputerem. Wiersz poleceń jest młodszym bratem języków programowania. Kiedy opanujesz pracę z wierszem poleceń, będziesz umiał sterować komputerem za pomocą języka. Gdy już będziesz miał to za sobą, możesz przejść do pisania kodu i poczuć się tak, jakbyś naprawdę władał tym kawałkiem złomu, który właśnie kupiłeś.

Jak korzystać z tego dodatku

Najlepszym sposobem korzystania z tego dodatku jest wykonanie następujących czynności.

- Weź mały notatnik i długopis.
- Zaczynij od początku dodatku i wykonaj każde ćwiczenie dokładnie tak, jak Ci powiedziano.
- Kiedy czytasz coś, co nie ma sensu lub czego nie rozumiesz, *zapisz to w swoim notatniku*. Zostaw trochę wolnego miejsca, aby później dopisać odpowiedź.
- Po zakończeniu ćwiczenia wróć do notatnika i przejrzyj zapisane pytania. Spróbuj na nie odpowiedzieć, wyszukując informacje w internecie i pytając znajomych, którzy mogą znać odpowiedź. Możesz napisać do mnie na adres help@learncodethehardway.org, a również pomogę.

Po prostu kontynuuj proces wykonywania ćwiczenia, zapisuj pytania, a potem wracaj do nich i odpowiadaj w miarę możliwości. Gdy skończysz, będziesz wiedzieć naprawdę dużo więcej o korzystaniu z wiersza poleceń niż Ci się wydaje.

Będziesz musiał zapamiętywać rzeczy

Ostrzegam Cię z wyprzedzeniem, że będę zmuszał do zapamiętywania od razu różnych rzeczy. To najszybszy sposób, aby posiadać jakąś umiejętność, ale dla niektórych ludzi zapamiętywanie jest bolesne. Po prostu walcz i rób to tak czy inaczej. Zapamiętywanie jest ważną umiejętnością podczas uczenia się różnych rzeczy, więc powinieneś pokonać swój lęk.

Oto sposób na zapamiętywanie.

- Powiedz sobie, że to *zrobisz*. Nie próbuj szukać wymówek i powodów, by się wykić, po prostu usiądź i rób to.
- Zapisz to, co chcesz zapamiętać, na fiszkach. Umieść połowę informacji na jednej stronie fiszki, a drugą połowę na odwrocie.
- Codziennie przez około 15 do 30 minut ćwicz z fiszkami, próbując przypomnieć sobie treść każdej z nich. Te fiszki, których nie udaje Ci się zgadnąć, umieść na osobnej kupce i po prostu ćwicz z nimi aż do znudzenia, a następnie spróbuj ponownie z całą talią i sprawdź, czy jest poprawa.
- Zanim położysz się spać, przez około 5 minut ćwicz z fiszkami, których nie odgadłeś prawidłowo.

Istnieją również inne techniki. Możesz na przykład spisać wszystko, czego potrzebujesz się nauczyć, na kartce papieru, zalaminować ją, a następnie przykleić do ściany pod prysznicem. Podczas kąpieli możesz ćwiczyć wiedzę bez patrzenia, a kiedy się zatniesz, zerkać na kartkę, aby odświeżyć sobie pamięć.

Jeśli będziesz robić to każdego dnia, powinieneś być w stanie zapamiętać większość rzeczy, których każę Ci się nauczyć na pamięć, mniej więcej w przedziale od tygodnia do miesiąca. Kiedy już to zrobisz, prawie wszystko inne stanie się łatwiejsze i bardziej intuicyjne, a to jest właśnie celem zapamiętywania. Nie chodzi o nauczanie się abstrakcyjnych koncepcji,

ale raczej o przyswojenie sobie podstaw, aby stały się dla Ciebie intuicyjne i żebyś nie musiał o nich myśleć. Gdy zapamiętasz te podstawy, przestaną być progami zwalniającymi, które uniemożliwiają naukę bardziej zaawansowanych abstrakcyjnych pojęć.

Konfiguracja

W tym dodatku będę chciał, abys zrobił trzy rzeczy. Oto one.

- Wykonaj pewne czynności w powłoce (wierszu poleceń, terminalu, programie PowerShell).
- Przeanalizuj, co właśnie zrobiłeś.
- Zrób więcej na własną rękę.

W pierwszym ćwiczeniu powinieneś znaleźć i uruchomić terminal, abys mógł wykonać resztę dodatku.

Zadanie

Uruchom terminal, powłokę lub program PowerShell, aby można było szybko uzyskać do nich dostęp i wiedzieć, że działają.

macOS

W systemie macOS musisz zrobić następujące kroki.

- Przytrzymaj przycisk *Command* i naciśnij spację.
- Pojawi się pasek wyszukiwania.
- Wpisz: **terminal**.
- Znalazona zostanie aplikacja Terminal, której ikona wygląda jak czarna skrzynka.
- Naciśnij *Enter*, aby otworzyć aplikację Terminal.
- Możesz teraz przejść do Docku i kliknąć prawym przyciskiem myszy, aby otworzyć menu kontekstowe, a następnie wybrać *Opcje/Zatrzymaj w Docku*.

Teraz masz otwarty terminal i jest on umieszczony w Twoim Docku, więc masz do niego łatwy dostęp.

Linux

Zakładam, że jeśli masz Linuksa, to już wiesz, jak uzyskać dostęp do terminala. Przejrzyj menu menedżera okien pod kątem wszystkiego o nazwie *Shell* lub *Terminal*.

Windows

W systemie Windows będziemy używać programu PowerShell. Niegdyś korzystano z programu o nazwie *cmd.exe*, ale nie on jest tak użyteczny jak PowerShell. Jeśli masz system Windows 7 lub nowszy, wykonaj następujące czynności.

- Kliknij *Start*.
- W pasku *Wyszukaj programy i pliki* wpisz **powershell**.
- Wciśnij *Enter*.

Jeżeli nie masz systemu w wersji co najmniej Windows 7, powinieneś *poważnie* rozważyć uaktualnienie. Jeśli jednak nadal nie chcesz aktualizować systemu do wyższej wersji, możesz spróbować zainstalować PowerShell z centrum pobierania Microsoftu. Wpisz w wyszukiwarce „powershell pobierz”, aby znaleźć aplikację dla Twojej wersji systemu Windows. Będziesz jednak zdany na siebie, ponieważ nie mam systemu Windows XP, ale mam nadzieję, że praca z programem PowerShell jest taka sama.

Czego się nauczyłeś

Nauczyłeś się, jak otworzyć terminal, abyś mógł wykonać pozostałe ćwiczenia z tego dodatku.

OSTRZEŻENIE! Jeśli masz takiego naprawdę inteligentnego przyjaciela, który zna już Linuksa, zignoruj go, kiedy każe Ci użyć czegoś innego niż bash. Uczę Cię powłoki bash. Tylko tyle! Twój przyjaciel będzie twierdził, że zsh da Ci dodatkowe 30 punktów IQ i wygrasz miliony na giełdzie. Zignoruj go. Twoim celem jest nabycie podstawowych umiejętności, a na tym poziomie nie ma znaczenia, której powłoki będziesz używać. Następne ostrzeżenie dotyczy unikania sieci IRC lub innych miejsc, w których spotykają się „hakerzy”. Sądzą oni, że zabawne jest pokazywanie Ci poleceń, które mogą zniszczyć Twój komputer. Polecenie `rm -rf /` jest klasycznym przykładem tego, czego *nigdy nie wolno wpisywać*. Po prostu ich unikaj. Jeśli potrzebujesz pomocy, upewnij się, że dostajesz ją od kogoś, komu ufasz, a nie od przypadkowych idiotów w internecie.

Zadanie dodatkowe

To ćwiczenie ma obszerny punkt „Zadanie dodatkowe”. Pozostałe ćwiczenia nie są tak angażujące jak to, ale chcę, żebyś przez to zadanie pamięciowe przygotował swój mózg na resztę dodatku. Zaufaj mi — dzięki temu później wiele rzeczy będzie łatwiejszych.

Linux i macOS

Wykorzystaj poniższą listę poleceń i przygotuj sobie fiszki z nazwami zapisanymi po jednej stronie i definicjami na odwrocie. Ćwicz je codziennie, kontynuując lekcje z tego dodatku.

`pwd` — wyświetla katalog roboczy
`hostname` — nazwa sieciowa komputera
`mkdir` — tworzy katalog
`cd` — zmienia katalog
`ls` — listuje zawartość katalogu
`rmdir` — usuwa katalog

pushd — umieszcza katalog na stosie
popd — zdejmuję katalog ze stosu
cp — kopiuje plik lub katalog
mv — przenosi plik lub katalog
less — przeglądanie zawartości pliku
cat — drukuje cały plik
xargs — wykonuje argumenty
find — znajduje pliki
grep — znajduje rzeczy wewnątrz plików
man — odczytuje stronę podręcznika
apropos — znajduje odpowiednią stronę podręcznika
env — sprawdza środowisko
echo — wypisuje kilka argumentów
export — eksportuje (ustawia) nową zmienną środowiskową
exit — wychodzi z powłoki
sudo — NIEBEZPIECZEŃSTWO! Stajesz się superużytkownikiem *root*. NIEBEZPIECZEŃSTWO!

Windows

Oto lista poleceń, jeśli używasz systemu Windows.

pwd — wyświetla katalog roboczy
hostname — nazwa sieciowa komputera
mkdir — tworzy katalog
cd — zmienia katalog
ls — listuje zawartość katalogu
rmdir — usuwa katalog
pushd — umieszcza katalog na stosie
popd — zdejmuję katalog ze stosu
cp — kopiuje plik lub katalog
robocopy — solidna kopia
mv — przenosi plik lub katalog
more — przeglądanie zawartości pliku
type — drukuje cały plik
forfiles — uruchamia polecenie dla wielu plików
dir -r — znajduje pliki

`select-string` — znajduje rzeczy wewnątrz plików

`help` — odczytuje stronę podręcznika

`helpctr` — znajduje odpowiednią stronę podręcznika

`echo` — wypisuje kilka argumentów

`set` — eksportuje (ustawia) nową zmienną środowiskową

`exit` — wychodzi z powłoki

`runas` — NIEBEZPIECZEŃSTWO! Stajesz się superużytkownikiem *root*. NIEBEZPIECZEŃSTWO!

Ćwicz, ćwicz, ćwicz! Wkuwaj dopóki, dopóty nie będziesz umiał wyrecytować tych definicji, gdy zobaczysz nazwę danego polecenia. Następnie ćwicz odwrotnie, abyś po przeczytaniu definicji potrafił wskazać, jakie polecenie robi daną rzecz. W ten sposób budujesz swoje słownictwo, ale nie poświęcaj na to aż tyle czasu, żeby dostać szału i się zniechęcić.

Ścieżki, foldery, katalogi (pwd)

W tym ćwiczeniu nauczysz się drukować katalog roboczy za pomocą polecenia `pwd`.

Zadanie

Nauczę Cię czytać te „sesje”, które będę pokazywał. Nie musisz wpisywać wszystkich rzeczy z listingów, tylko niektóre części.

- *Nie wpisujesz znaku \$ (w systemach uniksowych) lub > (w systemie Windows).* W ten sposób pokazuję Ci moją sesję, żebyś zobaczył, co tam mam.
- *Wpisujesz rzeczy po znaku \$ lub >, a następnie wciskasz Enter.* Jeśli więc widzisz \$ `pwd`, wpisujesz tylko `pwd` i wciskasz `Enter`.
- *Możesz wtedy zobaczyć moje dane wyjściowe, a po nich kolejny znak zachęty \$ lub >.* Ta treść to dane wyjściowe i powinieneś zobaczyć to samo u siebie.

Wpiszmy pierwsze proste polecenie, żebyś załapał, o co chodzi.

Linux i macOS

Ćwiczenie 2. — sesja

```
$ pwd
/Users/zedshaw
$
```

Windows

Ćwiczenie 2. — sesja Windows

```
PS C:\Users\zed> pwd
```

```
Path
```

```
----  
C:\Users\zed  
  
PS C:\Users\zed>
```

OSTRZEŻENIE! W tym dodatku muszę oszczędzać miejsce, abyś mógł skupić się na istotnych szczegółach poleceń. Zatem będę usuwał pierwszą część znaku zachęty (PS C:\Users\zed powyżej) i pozostawiał tylko niewielki fragment >. Oznacza to, że Twój znak zachęty nie będzie wyglądał dokładnie tak samo, ale nie przejmuj się. Pamiętaj, że od teraz będę miał tylko >, co będzie oznaczało cały znak zachęty wiersza poleceń. To samo zrobię dla znaku zachęty systemów uniksowych, ale w Uniksie znaki zachęty są tak różnorodne, że większość ludzi przyzwyczaiła się, iż \$ oznacza „po prostu znak zachęty”.

Czego się nauczyłeś

Twój znak zachęty będzie wyglądał inaczej niż mój. Przed znakiem \$ możesz mieć swoją nazwę użytkownika i nazwę Twojego komputera. W systemie Windows prawdopodobnie również będzie wyglądać to inaczej. Kluczowe jest to, że widzisz następujący wzorzec.

- Wyświetla się znak zachęty.
- Tam wpisujesz polecenie. W tym przypadku jest to pwd.
- Wydrukowane zostają pewne dane wyjściowe.
- Powtarzasz czynności.

Właśnie dowiedziałeś się, co robi polecenie pwd, którego pełna nazwa oznacza „wydrukuj katalog roboczy” (ang. *print working directory*). Co to jest katalog? Jest to folder. Folder i katalog są tym samym, a te pojęcia używane są zamiennie. Gdy otwierasz na komputerze eksplorator plików, aby graficznie wyszukiwać pliki, przeglądasz foldery. Foldery są dokładnie tym samym, co „katalogi”, z którymi będziemy pracować.

Zadanie dodatkowe

- Wpisz pwd 20 razy i za każdym razem powiedz na głos: „Wydrukuj katalog roboczy”
- Zapisz ścieżkę, którą podaje Ci to polecenie. Znajdź ją za pomocą wybranego przez Ciebie graficznego eksploratora plików.
- Naprawdę nie żartuję — wpisz to 20 razy i wypowiedz głośno. Bez marudzenia. Po prostu to zrób.

Jeśli się zgubisz

W trakcie wykonywania tych instrukcji możesz się zgubić. Możesz nie wiedzieć, gdzie jesteś i gdzie znajduje się plik, i na dodatek nie mieć pojęcia, co robić dalej. Aby rozwiązać ten problem, nauczę Cię poleceń, które powinieneś wpisać, żeby się odnaleźć.

Ilekcć się zgubisz, najprawdopodobniej będzie tak dlatego, że wpisywałeś polecenia i nie masz pojęcia, gdzie skończyłeś. Powinieneś wtedy wpisać `pwd`, aby *wydrukować bieżący katalog*. To podpowie Ci, gdzie jesteś.

Następną rzeczą, której potrzebujesz, jest sposób powrotu do bezpiecznego miejsca, czyli do katalogu głównego. W tym celu wpisz `cd ~` i już jesteś z powrotem w katalogu głównym.

Jeśli zatem zgubisz się w dowolnym momencie, wpisz następujące polecenia.

```
pwd
cd ~
```

Pierwsze polecenie `pwd` mówi Ci, gdzie jesteś. Drugie polecenie `cd ~` zabiera Cię do katalogu głównego, żebyś mógł spróbować ponownie.

Zadanie

Teraz za pomocą poleceń `pwd` i `cd ~` sprawdź, gdzie jesteś, a następnie wróć do katalogu głównego. W ten sposób upewnierz się, że zawsze będziesz we właściwym miejscu.

Czego się nauczyłeś

Nauczyłeś się, jak wrócić do katalogu głównego, jeśli kiedykolwiek się zgubisz.

Tworzenie katalogu (`mkdir`)

W tym ćwiczeniu dowiesz się, jak za pomocą polecenia `mkdir` utworzyć nowy katalog (folder).

Zadanie

Zapamiętaj! *Najpierw musisz wrócić do katalogu głównego!* Zanim wykonasz to ćwiczenie, wpisz swoje `pwd`, a następnie `cd ~`. Zanim wykonasz jakiegokolwiek ćwiczenie z tego dodatku, zawsze najpierw wróć do katalogu głównego!

Linux i macOS

Ćwiczenie 4. — sesja

```
$ pwd
$ cd ~
$ mkdir temp
$ mkdir temp/stuff
$ mkdir temp/stuff/things
$ mkdir -p temp/stuff/things/orange/apple/pear/grape
$
```

Windows

Ćwiczenie 4. — sesja Windows

```
> pwd
> cd ~
> mkdir temp
```

Directory: C:\Users\zed

Mode		LastWriteTime	Length	Name
----		-----	----	----
d----	12/17/2011	9:02 AM		temp

```
> mkdir temp/stuff
```

Directory: C:\Users\zed\temp

Mode		LastWriteTime	Length	Name
----		-----	----	----
d----	12/17/2011	9:02 AM		stuff

```
> mkdir temp/stuff/things
```

Directory: C:\Users\zed\temp\stuff

Mode		LastWriteTime	Length	Name
----		-----	----	----
d----	12/17/2011	9:03 AM		things

```
> mkdir temp/stuff/things/orange/apple/pear/grape
```

Directory: C:\Users\zed\temp\stuff\things\orange\apple\pear

Mode		LastWriteTime	Length	Name
----		-----	----	----
d----	12/17/2011	9:03 AM		grape

```
>
```

Jest to jedyny raz, kiedy wpiszę polecenia `pwd` i `cd ~`. Ty powinieneś je wpisywać w ćwiczeniach *za każdym razem*. Wpisuj je cały czas.

Czego się nauczyłeś

Teraz zaczynamy wpisywać więcej niż jedno polecenie. Wszystkie polecenia to różne sposoby uruchamiania `mkdir`. Co robi `mkdir`? Tworzy katalogi. Dlaczego o to pytasz? Powinieneś ćwiczyć Twoje fiszki i zapamiętywać polecenia. Jeśli nie wiesz, że „`mkdir` tworzy katalogi”, ćwicz dalej z fiszkami.

Co to znaczy utworzyć katalog? Możesz nazywać katalogi „folderami”. To to samo. Powyżej utworzyłeś po prostu katalogi wewnątrz katalogów wewnątrz kolejnych katalogów. Nazywa się to „ścieżką” i to jest sposób na powiedzenie „najpierw *temp*, potem *stuff*, następnie *things* i tu właśnie chcę być”. Jest to zestaw wskazówek dla komputera, który zawiera informację, że chcesz umieścić coś w określonym miejscu w drzewie folderów (katalogów) tworzących strukturę dysku twardego Twojego komputera.

OSTRZEŻENIE! W tym dodatku używam znaku / (prawego ukośnika) dla wszystkich ścieżek, ponieważ działa on teraz tak samo na wszystkich komputerach. Jednak użytkownicy systemu Windows powinni wiedzieć, że mogą również użyć znaku \ (lewego ukośnika), a niektórzy użytkownicy systemu Windows zwykle będą tego oczekiwać.

Zadanie dodatkowe

- Pojęcie „ścieżki” może być dla Ciebie w tym momencie mylące. Nie przejmuj się. Będziemy robić więcej ćwiczeń ze ścieżkami i wtedy to załapiesz.
- Utwórz 20 innych katalogów w katalogu *temp* na różnych poziomach. Przyjrzyj się im za pomocą graficznego eksploratora plików.
- Utwórz katalog ze spacją w nazwie, umieszczając tę nazwę w cudzysłowie, w ten sposób: `mkdir "To dobra zabawa"`.
- Jeśli katalog *temp* już istnieje, dostaniesz błąd. Użyj polecenia `cd`, aby przejść do katalogu roboczego, żebyś mógł spróbować tam. W systemie Windows dobrym miejscem jest *Pulpit*.

Zmienianie katalogu (cd)

W tym ćwiczeniu nauczysz się, jak zmieniać lokalizację z jednego katalogu na inny za pomocą polecenia `cd`.

Zadanie

Jeszcze raz podam Ci instrukcje dotyczące tych sesji.

- Nie wpisujesz znaku \$ (w systemach uniksowych) lub > (w systemie Windows).
- Wpisujesz wszystko, co jest po tym znaku i naciskasz *Enter*. Jeśli mam \$ `cd temp`, wpisujesz `cd temp` i naciskasz *Enter*.
- Dane wyjściowe pojawiają się po naciśnięciu *Enter*, a po nich następuje kolejny znak zachęty \$ lub >.
- Zawsze najpierw idź do katalogu głównego! Wpisz `pwd`, a następnie `cd ~`, aby powrócić do punktu początkowego.

Linux i macOS

Ćwiczenie 5. — sesja

```
$ cd temp
$ pwd
~/temp
$ cd stuff
$ pwd
~/temp/stuff
$ cd things
$ pwd
~/temp/stuff/things
$ cd orange/
$ pwd
~/temp/stuff/things/orange
$ cd apple/
$ pwd
~/temp/stuff/things/orange/apple
$ cd pear/
$ pwd
~/temp/stuff/things/orange/apple/pear
$ cd grape/
$ pwd
~/temp/stuff/things/orange/apple/pear/grape
$ cd ..
$ cd ..
$ pwd
~/temp/stuff/things/orange/apple
$ cd ..
$ cd ..
$ pwd
~/temp/stuff/things
$ cd ../../..
$ pwd
~/
$ cd temp/stuff/things/orange/apple/pear/grape
$ pwd
~/temp/stuff/things/orange/apple/pear/grape
$ cd ../../../../../../../
$ pwd
~/
$
```

Windows

Ćwiczenie 5. — sesja Windows

```
> cd temp
> pwd

Path
----
C:\Users\zed\temp
```

```
> cd stuff
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff
```

```
> cd things
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things
```

```
> cd orange
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things\orange
```

```
> cd apple
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things\orange\apple
```

```
> cd pear
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things\orange\apple\pear
```

```
> cd grape
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things\orange\apple\pear\grape
```

```
> cd ..
```

```
> cd ..
```

```
> cd ..
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things\orange
```

```
> cd ../..
```

```
> pwd
```

```
Path
----
C:\Users\zed\temp\stuff

> cd ..
> cd ..
> cd temp/stuff/things/orange/apple/pear/grape
> cd ../../../../../../../
> pwd

Path
----
C:\Users\zed

>
```

Czego się nauczyłeś

Utworzyłeś wszystkie katalogi w poprzednim ćwiczeniu, a teraz po prostu poruszasz się po nich za pomocą polecenia `cd`. W mojej sesji powyżej używam również `pwd`, aby sprawdzić, gdzie jestem, więc pamiętaj, żeby nie wpisywać danych wyjściowych z `pwd`. Przykładowo w linii 3. widzisz `~/temp`, ale są to dane wyjściowe z `pwd` ze znaku zachęty powyżej. *Nie wpisuj tego*.

Powinieneś także zobaczyć, w jaki sposób używam znaku `..`, aby przesunąć się „w górę” w drzewie i na ścieżce.

Zadanie dodatkowe

Bardzo ważną częścią nauki korzystania z interfejsu wiersza poleceń (ang. *command line interface* — **CLI**) na komputerze z graficznym interfejsem użytkownika (ang. *graphical user interface* — **GUI**) jest zrozumienie, w jaki sposób współpracują te dwa interfejsy. Kiedy zacząłem używać komputerów, nie było GUI, a wszystko robiło się w DOS-ie (CLI). Później, gdy komputery stały się na tyle potężne, że każdy mógł mieć grafikę, łatwo było mi dopasować katalogi CLI do okien i folderów GUI.

Jednak obecnie większość ludzi nie ma pojęcia o CLI, ścieżkach i katalogach. W rzeczywistości bardzo trudno ich tego nauczyć, a jedynym sposobem, aby zrozumieć ten związek, jest ciągła praca z CLI, aż pewnego dnia coś zaskoczy i rzeczy, które robisz w GUI, pojawią się w CLI.

W tym celu trzeba poświęcić nieco czasu na szukanie katalogów przy użyciu eksploratora plików GUI, a następnie odszukać je za pomocą CLI. Oto, co będziesz robić dalej.

- Używając `cd`, przejdź do katalogu `apple` za pomocą jednego polecenia.
- Używając `cd`, wróć do `temp` za pomocą jednego polecenia, ale nie idź wyżej.
- Dowiedz się, jak użyć `cd`, aby przejść do „katalogu głównego” za pomocą jednego polecenia.

- Używając `cd`, przejdź do katalogu *Dokumenty*, a następnie znajdź go za pomocą eksploratora plików GUI (takiego jak na przykład Finder, Windows Explorer czy podobny).
- Używając `cd`, przejdź do katalogu *Pobrane*, a następnie znajdź go w eksploratorze plików.
- Znajdź jakiś inny katalog za pomocą eksploratora plików, a następnie przejdź do niego, korzystając z `cd`.
- Pamiętaj, jak umieszczasz w cudzysłowie nazwę katalogu zawierającą spację? Możesz tak zrobić z każdym poleceniem. Jeśli masz na przykład katalog *To dobra zabawa*, możesz wpisać: `cd "To dobra zabawa"`.

Listowanie katalogu (`ls`)

W tym ćwiczeniu nauczysz się, jak wyświetlić zawartość katalogu za pomocą polecenia `ls`.

Zadanie

Zanim zaczniesz, użyj `cd`, aby przenieść się do katalogu powyżej `temp`. Jeśli nie masz pojęcia, gdzie jesteś, użyj `pwd`, aby to sprawdzić.

Linux / macOS

Ćwiczenie 6. — sesja

```
$ cd temp
$ ls
stuff
$ cd stuff
$ ls
things
$ cd things
$ ls
orange
$ cd orange
$ ls
apple
$ cd apple
$ ls
pear
$ cd pear
$ ls
$ cd grape
$ ls
$ cd ..
$ ls
grape
$ cd ../../../../
$ ls
orange
```

```
$ cd ../../
$ ls
stuff
$
```

Windows

Ćwiczenie 6. — sesja Windows

```
> cd temp
> ls
```

Directory: C:\Users\zed\temp

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		stuff

```
> cd stuff
> ls
```

Directory: C:\Users\zed\temp\stuff

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		things

```
> cd things
> ls
```

Directory: C:\Users\zed\temp\stuff\things

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		orange

```
> cd orange
> ls
```

Directory: C:\Users\zed\temp\stuff\things\orange

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		apple

```
> cd apple
> ls
```

Directory: C:\Users\zed\temp\stuff\things\orange\apple

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		pear

```
> cd pear
```

```
> ls
```

```
Directory: C:\Users\zed\temp\stuff\things\orange\apple\pear
```

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		grape

```
> cd grape
```

```
> ls
```

```
> cd ..
```

```
> ls
```

```
Directory: C:\Users\zed\temp\stuff\things\orange\apple\pear
```

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		grape

```
> cd ..
```

```
> ls
```

```
Directory: C:\Users\zed\temp\stuff\things\orange\apple
```

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		pear

```
> cd ../../..
```

```
> ls
```

```
Directory: C:\Users\zed\temp\stuff
```

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		things

```
> cd ..
```

```
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		stuff

```
>
```

Czego się nauczyłeś

Polecenie `ls` wyświetla zawartość katalogu, w którym aktualnie się znajdujesz. Pewnie zauważyłeś, że używam `cd`, aby przechodzić do różnych katalogów, a następnie wyświetlam ich zawartość, żeby wiedzieć, do którego katalogu przejść następnie.

Istnieje wiele opcji polecenia `ls`, ale nauczysz się, jak uzyskać pomoc w tej kwestii nieco później, kiedy będziemy omawiać polecenie `help`.

Zadanie dodatkowe

- *Wpisz każde z tych poleceń!* Musisz je naprawdę wpisać, aby się ich nauczyć. Samo czytanie nie wystarczy. Już przestaję krzyczeć.
- W systemach uniksowych spróbuj wpisać polecenie `ls -lR`, gdy jesteś w katalogu `temp`.
- W systemie Windows zrób to samo za pomocą `dir -R`.
- Użyj `cd`, aby przejść do innych katalogów na komputerze, a następnie użyj `ls` i zobacz, co się w nich znajduje.
- Zaktualizuj swój notatnik o nowe pytania. Wiem, że pewnie masz ich trochę, ponieważ nie omawiam wszystkich kwestii związanych z tym poleceniem.
- Pamiętaj, że jeśli się zgubisz, użyj `ls` i `pwd`, aby dowiedzieć się, gdzie jesteś, a następnie przejdź do wymaganej lokalizacji za pomocą `cd`.

Usuwanie katalogu (`rmdir`)

W tym ćwiczeniu dowiesz się, jak usunąć pusty katalog.

Zadanie

Linux i macOS

Ćwiczenie 7. — sesja

```
$ cd temp
$ ls
stuff
$ cd stuff/things/orange/apple/pear/grape/
$ cd ..
$ rmdir grape
$ cd ..
$ rmdir pear
$ cd ..
$ ls
apple
$ rmdir apple
$ cd ..
$ ls
orange
$ rmdir orange
$ cd ..
$ ls
things
$ rmdir things
```



```
$ cd ..
$ ls
stuff
$ rmdir stuff
$ pwd
~/temp
$
```

OSTRZEŻENIE! Jeśli spróbujesz wykonać polecenie `rmdir` w systemie macOS i dostaniesz odmowę usunięcia katalogu, chociaż będziesz *pewien*, że jest on pusty, to w rzeczywistości jest tam plik o nazwie `.DS_Store`. W takim przypadku wpisz `rm -rf <katalog>` (w miejsce `<katalog>` wstaw nazwę katalogu).

Windows

Ćwiczenie 7. — sesja Windows

```
> cd temp
> ls
```

Directory: C:\Users\zed\temp

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:03 AM		stuff

```
> cd stuff/things/orange/apple/pear/grape/
> cd ..
> rmdir grape
> cd ..
> rmdir pear
> cd ..
> rmdir apple
> cd ..
> rmdir orange
> cd ..
> ls
```

Directory: C:\Users\zed\temp\stuff

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/17/2011	9:14 AM		things

```
> rmdir things
> cd ..
> ls
```

Directory: C:\Users\zed\temp

Mode		LastWriteTime	Length	Name
------	--	---------------	--------	------

```
-----
d-----      12/17/2011    9:14 AM          stuff

> rmdir stuff
> pwd

Path
-----
C:\Users\zed\temp

> cd ..
>
```

Czego się nauczyłeś

Wymieszałem teraz różne polecenia, więc upewnij się, że wpisujesz je dokładnie, i bądź czujny. Za każdym razem, gdy popełniasz błąd, jest to spowodowane tym, że niewystarczająco się skupiasz. Jeśli okaże się, że robisz wiele błędów, zrób sobie przerwę lub po prostu zakończ na dziś. Zawsze jest jutro, żeby spróbować ponownie.

W tym przykładzie dowiedziałeś się, jak usunąć katalog. To łatwe. Po prostu przechodzisz do katalogu bezpośrednio powyżej, a następnie wpisujesz `rmdir <katalog>`, zastępując `<katalog>` nazwą katalogu do usunięcia.

Zadanie dodatkowe

- Utwórz 20 kolejnych katalogów i usuń je wszystkie.
- Utwórz pojedynczą ścieżkę katalogów, która ma 10 poziomów głębokości i usuwaj je pojedynczo, tak jak ją poprzednio.
- Jeśli spróbujesz usunąć katalog z zawartością, pojawi się błąd. Pokażę Ci, jak usuwać takie katalogi w późniejszych ćwiczeniach.

Poruszanie się po katalogach (pushd, popd)

W tym ćwiczeniu dowiesz się, jak zapisać bieżącą lokalizację i przejść do nowej lokalizacji za pomocą polecenia `pushd`. Następnie nauczysz się, jak wrócić do zapisanej lokalizacji za pomocą polecenia `popd`.

Zadanie

Linux i macOS

Ćwiczenie 8. — sesja

```
$ cd temp
$ mkdir i/like/icecream
$ pushd i/like/icecream
~/temp/i/like/icecream ~/temp
```

```

$ popd
~/temp
$ pwd
~/temp
$ pushd i/like
~/temp/i/like ~/temp
$ pwd
~/temp/i/like
$ pushd icecream
~/temp/i/like/icecream ~/temp/i/like ~/temp
$ pwd
~/temp/i/like/icecream
$ popd
~/temp/i/like ~/temp
$ pwd
~/temp/i/like
$ popd
~/temp
$ pushd i/like/icecream
~/temp/i/like/icecream ~/temp
$ pushd
~/temp ~/temp/i/like/icecream
$ pwd
~/temp
$ pushd
~/temp/i/like/icecream ~/temp
$ pwd
~/temp/i/like/icecream
$

```

Windows

Ćwiczenie 8. — sesja Windows

```

> cd temp
> mkdir i/like/icecream

```

Directory: C:\Users\zed\temp\i\like

Mode		LastWriteTime	Length	Name
----		-----	-----	----
d----	12/20/2011	11:05 AM		icecream

```

> pushd i/like/icecream
> popd
> pwd

```

```

Path
----
C:\Users\zed\temp

```

```

> pushd i/like
> pwd

```

```
Path
----
C:\Users\zed\temp\i\like

> pushd icecream
> pwd

Path
----
C:\Users\zed\temp\i\like\icecream

> popd
> pwd

Path
----
C:\Users\zed\temp\i\like

> popd
>
```

OSTRZEŻENIE! W systemie Windows zazwyczaj nie potrzebujesz opcji `-p`, tak jak w Linuksie. Uważam jednak, że jest to nowsza wersja, więc możesz trafić na starsze wersje PowerShell w systemie Windows, które wymagają opcji `-p`. Jeśli masz więcej informacji na ten temat, proszę napisz do mnie na adres help@learncodethehardway.org, żebym mógł wyjaśnić wątpliwości, czy wspomnieć o opcji `-p` w systemie Windows, czy nie.

Czego się nauczyłeś

Wykonując te polecenia, wkraczasz na terytorium programisty, ale są one tak przydatne, że muszę Cię ich nauczyć. Polecenia pozwalają Ci tymczasowo przejść do innego katalogu, a następnie wrócić, łatwo przełączając się między tymi dwoma katalogami.

Polecenie `pushd` pobiera bieżący katalog i „umieszcza” go na liście do późniejszego użycia, a następnie zmienia lokalizację na inny katalog. To tak, jakby powiedzieć: „Zapisz, gdzie jestem, a potem idź tutaj”.

Polecenie `popd` bierze ostatni zapisany katalog i „zdejmuje” go z listy, zabierając Cię z powrotem w to miejsce.

Ostatnia uwaga dotyczy polecenia `pushd` w systemach uniksowych. Jeśli uruchomisz je samo bez żadnych argumentów, przełączy Cię pomiędzy Twoim aktualnym katalogiem i ostatnim, który zapisałeś za pomocą `pushd`. Jest to łatwy sposób przełączania się między dwoma katalogami. *Jednak nie działa w PowerShell.*

Zadanie dodatkowe

- Użyj tych poleceń, aby poruszać się po katalogach w całym komputerze.
- Usuń katalogi *i/like/icecream* i utwórz własne. Następnie pochodź po nich.
- Wyjaśnij sam sobie dane wyjściowe, które otrzymasz z `pushd` i `popd`. Zauważyłeś, że to działa jak stos?
- Już to wiesz, ale pamiętaj, że `mkdir -p` (w systemach Linux i macOS) utworzy całą ścieżkę, nawet jeśli żaden z katalogów nie istnieje. To właśnie zrobiłem na początku w tym ćwiczeniu.
- Pamiętaj, że system Windows utworzy pełną ścieżkę i nie potrzebuje opcji `-p`.

Tworzenie pustych plików (touch/New-Item)

W tym ćwiczeniu nauczysz się, jak utworzyć pusty plik za pomocą polecenia `touch` (New-Item w systemie Windows).

Zadanie

Linux i macOS

Ćwiczenie 9. — sesja

```
$ cd temp
$ touch iamcool.txt
$ ls
iamcool.txt
$
```

Windows

Ćwiczenie 9. — sesja Windows

```
> cd temp
> New-Item iamcool.txt -type file
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/17/2011 9:03 AM		iamcool.txt

```
>
```

Czego się nauczyłeś

Nauczyłeś się, jak utworzyć pusty plik. W systemach uniksowych robi to polecenie `touch`, które zmienia również datę modyfikacji pliku. Rzadko używam go do czegoś innego niż tworzenie pustych plików. W systemie Windows nie masz tego polecenia, więc nauczyłeś się korzystać z polecenia `New-Item`, które robi to samo, ale może również tworzyć nowe katalogi.

Zadanie dodatkowe

- W systemie uniksowym utwórz katalog, przejdź do niego, a następnie utwórz w nim plik. Potem przejdź jeden poziom wyżej i uruchom polecenie `rmdir` w tym katalogu. *Powinieneś* otrzymać błąd. Spróbuj zrozumieć, dlaczego dostajesz ten błąd.
- W systemie Windows zrób to samo, ale nie dostaniesz błędu. Otrzymasz pytanie, czy naprawdę chcesz usunąć katalog.

Kopiowanie pliku (`cp`)

W tym ćwiczeniu nauczysz się kopiować plik z jednej lokalizacji do drugiej za pomocą polecenia `cp`.

Zadanie

Linux i macOS

Ćwiczenie 10. — sesja

```
$ cd temp
$ cp iamcool.txt neat.txt
$ ls
iamcool.txt neat.txt
$ cp neat.txt awesome.txt
$ ls
awesome.txt iamcool.txt neat.txt
$ cp awesome.txt thefourthfile.txt
$ ls
awesome.txt iamcool.txt neat.txt thefourthfile.txt
$ mkdir something
$ cp awesome.txt something/
$ ls
awesome.txt iamcool.txt neat.txt something thefourthfile.txt
$ ls something/
awesome.txt
$ cp -r something newplace
$ ls newplace/
awesome.txt
$
```

Windows

Ćwiczenie 10. — sesja Windows

```
> cd temp
> cp iamcool.txt neat.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt

```
> cp neat.txt awesome.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
-a---	12/22/2011 4:49 PM	0	awesome.txt
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt

```
> cp awesome.txt thefourthfile.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
-a---	12/22/2011 4:49 PM	0	awesome.txt
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt
-a---	12/22/2011 4:49 PM	0	thefourthfile.txt

```
> mkdir something
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
d----	12/22/2011 4:52 PM		something

```
> cp awesome.txt something/
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
d----	12/22/2011 4:52 PM		something
-a---	12/22/2011 4:49 PM	0	awesome.txt

```
-a---      12/22/2011   4:49 PM      0 iamcool.txt
-a---      12/22/2011   4:49 PM      0 neat.txt
-a---      12/22/2011   4:49 PM      0 thefourthfile.txt
```

```
> ls something
```

```
Directory: C:\Users\zed\temp\something
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/22/2011 4:49 PM	0	awesome.txt

```
> cp -recurse something newplace
```

```
> ls newplace
```

```
Directory: C:\Users\zed\temp\newplace
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/22/2011 4:49 PM	0	awesome.txt

```
>
```

Czego się nauczyłeś

Teraz możesz kopiować pliki. Łatwo po prostu wziąć plik i skopiować go do nowego pliku. W tym ćwiczeniu tworzę również nowy katalog i kopiuję plik do tego katalogu.

Zdradzę Ci teraz sekret o programistach i administratorach systemów. Są leniwi. Ja też jestem leniwy. Moi przyjaciele są leniwi. Dlatego używamy komputerów. Lubimy, żeby komputery robiły za nas nudne rzeczy. W dotychczasowych ćwiczeniach wpisywałeś powtarzające się nudne polecenia, żebyś mógł się ich nauczyć, ale zazwyczaj tak nie jest. Zwykle jeśli robisz coś nudnego i powtarzalnego, prawdopodobnie znajdzie się taki programista, który wpadł już na pomysł, jak to ułatwić. Po prostu jeszcze o tym nie wiesz.

Kolejną rzeczą, która dotyczy programistów, jest to, że nie są nawet w połowie tak mądrzy, jak Ci się wydaje. Jeśli będziesz za bardzo zastanawiał się, co wpisać, prawdopodobnie zrobisz to źle. Zamiast tego spróbuj sobie wyobrazić, jak może nazywać się polecenie i spróbuj wpisać. Istnieje prawdopodobieństwo, że będzie to nazwa, którą wymyśliłeś lub podobny do niej skrót. Jeśli mimo wszystko nie uda Ci się wpaść na to intuicyjnie, popytaj innych i poszukaj w internecie. Można tylko mieć nadzieję, że nie będzie to coś naprawdę głupiego, takiego jak ROBOCOPY.

Zadanie dodatkowe

- Użyj polecenia `cp -r`, aby skopiować więcej katalogów wraz ze znajdującymi się w nich plikami.
- Skopiuj plik do katalogu głównego lub na pulpit.
- Znajdź te pliki w GUI i otwórz je w edytorze tekstów.

- Zwróciłeś uwagę, że czasami umieszczam / (prawy ukośnik) na końcu katalogu? W ten sposób upewniam się, że plik jest naprawdę katalogiem, więc jeśli katalog nie istnieje, dostanę błąd.

Przenoszenie pliku (mv)

W tym ćwiczeniu nauczysz się przenosić plik z jednej lokalizacji do drugiej za pomocą polecenia `mv`.

Zadanie

Linux i macOS

Ćwiczenie 11. — sesja

```
$ cd temp
$ mv awesome.txt uncool.txt
$ ls
newplace uncool.txt
$ mv newplace oldplace
$ ls
oldplace uncool.txt
$ mv oldplace newplace
$ ls
newplace uncool.txt
$
```

Windows

Ćwiczenie 11. — sesja Windows

```
> cd temp
> mv awesome.txt uncool.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
d----	12/22/2011 4:52 PM		newplace
d----	12/22/2011 4:52 PM		something
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt
-a---	12/22/2011 4:49 PM	0	thefourthfile.txt
-a---	12/22/2011 4:49 PM	0	uncool.txt

```
> mv newplace oldplace
> ls
```

Directory: C:\Users\zed\temp

```

Mode                LastWriteTime         Length Name
----                -
d----             12/22/2011     4:52 PM             oldplace
d----             12/22/2011     4:52 PM             something
-a---             12/22/2011     4:49 PM             0 iamcool.txt
-a---             12/22/2011     4:49 PM             0 neat.txt
-a---             12/22/2011     4:49 PM             0 thefourthfile.txt
-a---             12/22/2011     4:49 PM             0 uncool.txt

```

```
> mv oldplace newplace
```

```
> ls newplace
```

```
Directory: C:\Users\zed\temp\newplace
```

```

Mode                LastWriteTime         Length Name
----                -
-a---             12/22/2011     4:49 PM             0 awesome.txt

```

```
> ls
```

```
Directory: C:\Users\zed\temp
```

```

Mode                LastWriteTime         Length Name
----                -
d----             12/22/2011     4:52 PM             newplace
d----             12/22/2011     4:52 PM             something
-a---             12/22/2011     4:49 PM             0 iamcool.txt
-a---             12/22/2011     4:49 PM             0 neat.txt
-a---             12/22/2011     4:49 PM             0 thefourthfile.txt
-a---             12/22/2011     4:49 PM             0 uncool.txt

```

```
>
```

Czego się nauczyłeś

Nauczyłeś się przenosić pliki lub raczej zmieniać ich nazwę. To proste: podajesz starą nazwę i nową nazwę.

Zadanie dodatkowe

Przenieś plik z katalogu *newplace* do innego katalogu, a następnie przenieś go z powrotem.

Przeglądanie pliku (less/more)

Aby wykonać to ćwiczenie, popracujesz trochę z poleceniami, które już znasz. Będziesz także potrzebował edytora tekstu, który może tworzyć zwykłe pliki tekstowe (.txt). Oto co zrobisz.

- Otwórz edytor tekstu i wpisz jakiś tekst w nowym pliku. W systemie macOS może to być edytor tekstu TextWrangler. W systemie Windows może to być Notepad++. W systemie Linux może to być gedit. Każdy edytor się nada.

- Zapisz ten plik na pulpicie i nadaj mu nazwę *test.txt*.
- W powłoce użyj poleceń, które znasz, aby *skopiować* ten plik do katalogu *temp*, z którym do tej pory pracowałeś.

Gdy to zrobisz, wykonaj ćwiczenie.

Zadanie

Linux i macOS

Ćwiczenie 12. — sesja

```
$ less test.txt
[tu wyświetla się zawartość pliku]
$
```

To wszystko! Aby wyjść z *less*, wpisz *q* (ang. *quit*).

Windows

Ćwiczenie 12. — sesja Windows

```
> more test.txt
[tu wyświetla się zawartość pliku]
>
```

OSTRZEŻENIE! W powyższych danych wyjściowych umieściłem informację [*tu wyświetla się zawartość pliku*], aby „skrócić” to, co pokazuje ten program. Robię tak, gdy chcę powiedzieć: „Pokazanie Ci danych wyjściowych z tego programu jest zbyt skomplikowane, więc po prostu wstaw to, co widzisz na swoim komputerze, i udawaj, że ja Ci to pokazałem”. Twój ekran naprawdę nie wyświetli tej informacji.

Czego się nauczyłeś

To jeden ze sposobów sprawdzenia zawartości pliku. Jest to przydatne, ponieważ jeśli plik ma wiele linii, zostanie podzielony na „strony”, więc widoczny będzie tylko jeden pełny ekran danych. W punkcie „Zadanie dodatkowe” pobawisz się z tym trochę więcej.

Zadanie dodatkowe

- Ponownie otwórz swój plik tekstowy i wielokrotnie kopiuj oraz wklejaj tekst, aby plik miał około 50 – 100 linii długości.
- Skopiuj go ponownie do katalogu *temp*, żebyś mógł go podejrzeć.

- Teraz wykonaj ponownie to ćwiczenie, ale tym razem przewijaj strony. W systemach uniksowych używasz spacji i litery *w*, aby przewijać w dół i w górę. Klawisze strzałek również działają. W systemie Windows wystarczy nacisnąć spację, aby przewijać strony.
- Podejrzyj także kilka pustych plików, które utworzyłeś.
- Polecenie `cp` nadpisze pliki, które już istnieją, więc należy zachować ostrożność podczas kopiowania plików.

Strumieniowanie pliku (`cat`)

Do tego ćwiczenia musisz jeszcze dodać trochę konfiguracji, więc przyzwyczaisz się do tworzenia plików w jednym programie, a następnie uzyskiwania do nich dostępu z wiersza poleceń. Korzystając z tego samego edytora tekstu, co poprzednio, utwórz kolejny plik o nazwie `test2.txt`, ale tym razem zapisz go bezpośrednio w swoim katalogu `temp`.

Zadanie

Linux i macOS

Ćwiczenie 13. — sesja

```
$ less test2.txt
[tu wyświetla się zawartość pliku]
$ cat test2.txt
Jestem fajny gość.
Powiedziałby ktoś?
I takie piszę wiersze,
że laseczki mają dość.
$ cat test.txt
Cześć, to jest czaderskie.
$
```

Windows

Ćwiczenie 13. — sesja Windows

```
> more test2.txt
[tu wyświetla się zawartość pliku]
> cat test2.txt
Jestem fajny gość.
Powiedziałby ktoś?
I takie piszę wiersze,
że laseczki mają dość.
> cat test.txt
Cześć, to jest czaderskie.
>
```

Pamiętaj, że kiedy mówię `[tu wyświetla się zawartość pliku]`, skracam dane wyjściowe z tego polecenia, żeby nie musiał pokazywać Ci wszystkiego dokładnie.

Czego się nauczylesz

Podoba Ci się mój wiersz? Na pewno dostanę Nobla. W każdym razie znasz już pierwsze polecenie i w ten sposób po prostu sprawdzam, czy ten plik tam jest. Potem za pomocą polecenia `cat` puszczam strumień pliku na ekran. To polecenie po prostu wyrzuca cały plik na ekran bez stronicowania lub zatrzymywania go. Aby to udowodnić, muszę to zrobić z plikiem *test.txt*, który powinien wypłuć wiele linii tekstu.

Zadanie dodatkowe

- Utwórz kilka dodatkowych plików tekstowych i popracuj z poleceniem `cat`.
- W systemach uniksowych spróbuj wpisać `cat test.txt test2.txt` i zobacz, co się stanie.
- W systemie Windows spróbuj wpisać `cat test.txt, test2.txt` i zobacz, co się stanie.

Usuwanie pliku (rm)

W tym ćwiczeniu dowiesz się, jak usunąć (wykasować) plik za pomocą polecenia `rm`.

Zadanie

Linux

Ćwiczenie 14. — sesja

```
$ cd temp
$ ls
uncool.txt iamcool.txt neat.txt something thefourthfile.txt
$ rm uncool.txt
$ ls
iamcool.txt neat.txt something thefourthfile.txt
$ rm iamcool.txt neat.txt thefourthfile.txt
$ ls
something
$ cp -r something newplace
$$
rm something/awesome.txt
$ rmdir something
$ rm -rf newplace
$ ls
$
```

Windows

Ćwiczenie 14. — sesja Windows

```
> cd temp
> ls
```

Directory: C:\Users\zed\temp

Mode		LastWriteTime	Length	Name
d----	12/22/2011	4:52 PM		newplace
d----	12/22/2011	4:52 PM		something
-a---	12/22/2011	4:49 PM	0	iamcool.txt
-a---	12/22/2011	4:49 PM	0	neat.txt
-a---	12/22/2011	4:49 PM	0	thefourthfile.txt
-a---	12/22/2011	4:49 PM	0	uncool.txt

```
> rm uncool.txt
> ls
```

Directory: C:\Users\zed\temp

Mode		LastWriteTime	Length	Name
d----	12/22/2011	4:52 PM		newplace
d----	12/22/2011	4:52 PM		something
-a---	12/22/2011	4:49 PM	0	iamcool.txt
-a---	12/22/2011	4:49 PM	0	neat.txt
-a---	12/22/2011	4:49 PM	0	thefourthfile.txt

```
> rm iamcool.txt
> rm neat.txt
> rm thefourthfile.txt
> ls
```

Directory: C:\Users\zed\temp

Mode		LastWriteTime	Length	Name
d----	12/22/2011	4:52 PM		newplace
d----	12/22/2011	4:52 PM		something

```
> cp -r something newplace
> rm something/awesome.txt
> rmdir something
> rm -r newplace
> ls
>
```

Czego się nauczyłeś

Tutaj czytamy pliki z ostatniego ćwiczenia. Pamiętasz, kiedy kazałem Ci wypróbować polecenie `rmdir` na katalogu, w którym coś było? Cóż, nie udało się, ponieważ nie można usunąć katalogu zawierającego pliki. Aby to zrobić, musisz usunąć plik lub rekursywnie usunąć całą jego zawartość. To właśnie zrobiłeś na końcu tego ćwiczenia.

Zadanie dodatkowe

- Wyczyść całą zawartość *temp* ze wszystkich ćwiczeń, które wykonałeś do tej pory.
- Zapisz w swoim notatniku, aby zachować ostrożność podczas uruchamiania rekursywnego usuwania plików.

Wyjście z terminala (exit)

Zadanie

Linux i macOS

Ćwiczenie 15. — sesja

```
$ exit
```

Windows

Ćwiczenie 15. — sesja Windows

```
> exit
```

Czego się nauczyłeś

Ostatnim ćwiczeniem jest wyjście z terminala. Ponownie jest to bardzo łatwe, ale mam dla Ciebie więcej zadań.

Zadanie dodatkowe

Twoim ostatnim zestawem ćwiczeń będzie użycie systemu pomocy, aby sprawdzić zestaw poleceń, które samodzielnie powinieneś zbadać i których musisz nauczyć się używać.

Oto lista dla systemów uniksowych:

- `xargs`,
- `sudo`,
- `chmod`,
- `chown`.

W systemie Windows sprawdź następujące rzeczy:

- forfiles,
- runas,
- attrib,
- icacls.

Dowiedz się, co to za polecenia, pobaw się z nimi, a następnie dodaj je do swoich fiszek.

Następne kroki z wierszem poleceń

Ukończyłeś kurs przyspieszony. W tym momencie jesteś ledwo początkującym użytkownikiem powłoki. Istnieje cała ogromna lista trików i kluczowych sekwencji, których jeszcze nie znasz, a ja wskażę Ci kilka zasobów, w których możesz poszukać dalszych informacji.

Zasoby dla uniksowej powłoki bash

Powłoka, której używasz, nazywa się bash. Nie jest to najlepsza powłoka, ale jest wszechobecna i ma dużo funkcji, więc to dobry początek. Oto krótka lista linków na temat powłoki bash, z którymi powinieneś się zapoznać.

Linux Bash Shell Cheat Sheet

https://learncodethehardway.org/unix/bash_cheat_sheet.pdf

(autor: Raphael, na licencji CC).

Bash Reference Manual <http://www.gnu.org/software/bash/manual/bashref.html>.

Zasoby dla programu PowerShell

W systemie Windows jest naprawdę tylko program PowerShell. Oto lista przydatnych linków związanych z PowerShell.

Windows PowerShell Owner's Manual <http://technet.microsoft.com/en-us/library/ee221100.aspx>.

Ściągawka z PowerShell https://download.microsoft.com/download/2/1/2/2122F0B9-0EE6-4E6D-BFD6-F9DCD27C07F9/WS12_QuickRef_Download_Files/PowerShell_LangRef_v3.pdf.

Master PowerShell <http://powershell.com/cs/blogs/ebook/default.aspx>.

Skorowidz

A

adres URL, 247, 249
 przekazywanie parametrów, 249
 algebra Boole'a, 120
 alternatywa, 119, 151
 argument wiersza poleceń, 64, 65, 66
 ASCII, 104, 105
 Atom, 20, 23, 25
 atrybut, 174

B

bajt, 104
 biblioteka, 65
 bit, 104
 blok
 except, 228
 try, 228
 try-except, 231
 błąd, 30
 EOL while scanning string literal, 82
 importu, 231
 invalid syntax, 69
 komunikat, 31
 not enough values to unpack,, 66
 not enough values to unpack., 70
 parsowania, 233

C

CLI, 284
 command line interface, *Patrz:* CLI

D

dane
 struktura, *Patrz:* struktura danych
 wprowadzanie, 226
 DBES, 107
 debugger, 149
 dictionary, *Patrz:* słownik
 działanie kolejność wykonywania, 38
 dziecko, *Patrz:* klasa potomna

dziedziczenie, 174, 200, 203, 205, 207
 domyślne, 201
 wielokrotne, 200, 205, 207
 dzwonek, 152

E

edytor tekstu, 25, 76
 Atom, *Patrz:* Atom
 Emacs, 25
 Vim, 25
 escape sequence, *Patrz:* sekwencja ucieczki

F

format string, *Patrz:* łańcuch znaków
 sformatowany
 formularz, 246, 248, 249
 framework
 flask, 238, 241, 253
 dziennik, 239
 instalowanie, 238
 tryb debuggera, 241
 nose, 215, 219
 testowy, 215
 webowy, 238
 f-string, 46, 152
 funkcja, 84, 186
 __init__, 205, 211
 anonimowa, 151
 assert_equal, 224
 assert_raises, 237
 die, 148
 format, 46, 52
 input, 61, 62, 68
 int, 228
 join, 160
 len, 82
 liczba argumentów, 90
 lista kontrolna, 86
 nadpisanie, 202
 nazwa, 86, 211
 range, 132, 136, 160
 readline, 94
 round, 45

funkcja
 super, 202, 203, 205
 tworzenie, 84, 211
 wartość, 96
 wywoływanie, 89

H

HTML, 242, 249

I

IDE, 25
IDLE, 26
importowanie, 64
instancja, 170, 174, 180
instrukcja
 elif, 129, 130, 131, 150
 else, 129, 130, 150
 if, 126, 127, 128, 130, 131, 151
 if-else, 131
Integrated Development Environment,
 Patrz: IDE
interfejs API, 253

J

język, 105
 programowania, 270
 JavaScript, 208, 267
 nauka, 267
 obiektowego, 168
 prototypowy, 208
 Ruby, 136, 267

K

catalog
 listowanie, 285
 tworzenie, 279
 usuwanie, 288
 zmiana, 281
klasa, 150, 168, 169, 171, 174, 186, 208
 bazowa, 192, 201
 hierarchia, 186, 188
 nadrzędna, 200
 nazwa, 211
 potomna, 201
 tworzenie, 189

kodek, 104
kodowanie, 104
kolejność
 rozwiązywania metod, *Patrz:* MRO
 wykonywania działań, 38
komentarz, 34, 212
 ## ??, 184
 dokumentujący, 114
kompozycja, 174, 207
koniunkcja, 119, 150
krotka, 227, 232

L

liczba
 całkowita, 152
 dziesiętna, 152
 jako łańcuch znaków, 61
 kardynalna, 142, 159
 konwertowanie, 227
 ósemkowa, 152
 szesnastkowa, 152
 zmiennoprzecinkowa, 37, 41, 152
 zaokrąglenie, 45
lista, 134, 136, 156, 157, 158, 166
 dwuwymiarowa, 136
 element, 142
 kopiowanie, 178
 tworzenie, 134
 wycinek, 178
 zastosowania, 159
logika, 118
 boolowska, 122

Ł

łańcuch znaków, 35, 44, 46, 102, 107, 157
 formatowanie, 46, 52, 152
 kod formatowania, 152
 literał sformatowany, 46
 sformatowany, 44
 z osadzonymi zmiennymi, 44

M

method resolution order, *Patrz:* MRO
metoda, 211
 post, 253

moduł, 65, 168, 169, 207
doctest, 225
MRO, 205

N

negacja, 119, 151
notacja camelCase, 211

O

obiekt, 170, 174, 181, 186, 208
plik, 74
operacja
append, 136
modulo, 37
operator, 119, 153
inkrementacji, 127
logiczny, 119
oprogramowania konfiguracja, 214, 215

P

Parentheses Exponents Multiplication
Division Addition Subtraction, *Patrz:*
łańcuch znaków
parser, 233, 236
PEMDAS, 38
pętla
for, 134, 140, 148, 150, 151
while, 138, 140, 148, 151, 160, 191
plik, 93
.py, 64
HTML, 242, 249
kasowanie, 301
konkatenacja, 81
kopiowanie, 80, 296
languages.txt, 102
modyfikator trybu, 78
odczyt, 76
opróżnianie, 76
otwieranie, 72, 74
przenoszenie, 297
pusty, 293
tekstowy, 72, 298
tryb, 78
zamykanie, 76
zapisywanie, 76
podpowiedź, 62

polecenie
apropos, 276
cat, 81, 276, 300
cd, 275, 276, 279, 281
close, 76
cp, 276, 294
dir, 276
echo, 81, 276, 277
env, 276
exit, 146, 276, 277, 303
export, 276
find, 276
forfiles, 276
format, 52
grep, 276
help, 277
helpctr, 277
hostname, 275, 276
import, 80
less, 276, 298, 299
ls, 275, 276, 285, 287
man, 276
mkdir, 275, 276, 279, 280, 293
more, 276, 298
mv, 276, 297
New-Item, 293
nosetests, 223, 224, 225
open, 72
pip, 214, 215
pip3.6, 214
popd, 276, 290, 292
print, 28, 115, 149
pushd, 276, 290, 292
pwd, 275, 276, 277, 278, 279
pydoc, 63
pydoc input, 62
python, 219
quit, 114
read, 76
readline, 76
return, 115
rm, 301
rmdir, 275, 276, 288, 289
robocopy, 276
runas, 277
seek, 76, 93, 94
select-string, 277
set, 277
setuptools, 214
sudo, 276

polecenie
 touch, 293
 truncate, 76, 78
 type, 276
 virtualenv, 214, 215
 write, 76
 xargs, 276
PowerShell, 20, 21, 30, 274, 304
powłoka, 274
 bash, 20, 304
programowanie, 270
 obiektywne, 159, 168, 186, 200
 z dołu do góry, 191
 z góry na dół, 186, 191
projekt szkielet, 214, 217, 218, 220
protokół, 247
przeglądarka, 239, 240, 247

R

refaktoryzacja, 256
rodzic, *Patrz:* klasa nadrzędna

S

sekwencja ucieczki, 56, 57, 152
serwer, 248, 254
składnia .format, 52
skrypt, 64
słownik, 150, 162, 165, 166, 168, 169
 mapowanie, 163, 165
słowo kluczowe
 and, 150
 as, 150
 assert, 150
 break, 150
 class, 150, 174
 continue, 150
 def, 84, 150
 del, 150, 163
 elif, 150
 else, 150
 except, 150, 151, 228
 exec, 150
 False, 53, 119
 finally, 150
 for, 150
 from, 150
 global, 150
 if, 151

import, 151
in, 151
is, 151
lambda, 151
not, 151
or, 151
pass, 151
print, 151
raise, 151, 233
return, 96, 151
True, 53, 119
try, 151, 228
while, 151
with, 151
yield, 151

standard

 ASCII, *Patrz:* ASCII

 Unicode, 105, 106

string, *Patrz:* łańcuch znaków

struktura danych, 158, 165

 collections.OrderedDict, 166

system

 Big5, 109

 UTF-8, 102, 106

szkielet, 214, 217, 218, 220

Ś

ścieżka, 281

środowisko programistyczne

 virtualenv, 214

 zintegrowane, *Patrz:* IDE

T

tablica, 136

 prawdy, 119, 122

tabulator, 152

terminal, 20, 23, 29, 274

test

 dla formularza, 252

 fałszywa sesja, 264

 jednostkowy, 222, 227

 uruchomienie, 223, 224, 225

 zautomatyzowany, 222

token błędu, 227

typ

 bytes, 102, 151

 danych, 151

dicts, 151
 False, 151
 floats, 151
 lists, 151
 None, 151
 numbers, 151
 string, 102
 strings, 151
 True, 151

W

wiersz poleceń, 63, 272
 argument, *Patrz:* argument wiersza
 polecień
 interfejsu, *Patrz:* CLI
 wyjątek
 obsługa, 228, 233
 ValueError, 228

Z

zmienna, 40
 argumentów, 64
 argv, 64
 globalna, 90, 150, 211
 nazwa, 45
 skrótowa, 46
 w funkcji, 88
 w skrypcie, 88
 znak
 !=, *Patrz:* operator logiczny !=
 " \t, 56
 #, *Patrz:* znak kratki
 %, 36, 37
 %%%, 153
 %c, 153
 %d, 152
 %e, 152
 %E, 152
 %f, 152
 %F, 152
 %g, 152
 %G, 152
 %i, 152
 %o, 152
 %r, 153
 %s, 153
 %u, 152

%x, 152
 %X, 152
 /, *Patrz:* znak ukośnika prawego
 ^, *Patrz:* znak wstawiania
 _, *Patrz:* znak podkreślenia
 {}, 44
 +=, 94, 127
 <=, 119
 =, *Patrz:* znak równości
 ==, 119, *Patrz:* znak równości
 podwójny
 >, *Patrz:* znak zachęty
 >=, 119
 \a, 152
 ASCII, *Patrz:* ASCII
 \b, *Patrz:* znak backspace
 backspace, 152
 cudzysłowu
 podwójnego, 31, 46, 48, 51, 54, 55,
 56, 152
 pojedynczego, 46, 48, 51, 56, 152
 \f, 152
 gwiazdka, 86
 kratki, 31, 34, 35
 łańcuch, *Patrz:* łańcuch znaków
 \n, *Patrz:* znak nowej linii
 nawias
 klamrowy, 62
 okrągły, 62
 nowej linii, 56, 152
 podkreślenia, 40
 powrotu karetki, 152
 \r, *Patrz:* znak powrotu karetki
 równości, 42, 96
 podwójny, 42
 \t, 152
 ukośnika
 lewego, 56, 58, 152, 281
 prawego, 58, 281
 \v, 152
 wstawiania, 31
 zachęty, 68, 113, 146, 278

Ż

żądanie, 246, 247
 odpowiedź, 248
 POST, 253

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>