

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Python. Wprowadzenie. Wydanie III

Autor: Mark Lutz

Tłumaczenie: Anna Trojan

ISBN: 978-83-246-1648-0

Tytuł oryginału: [Learning Python, 3rd Edition](#)

Format: 168x237, stron: 736



Poznaj i wykorzystaj w praktyce aktualne możliwości Pythona

- Jak Python wykonuje programy?
- Jak korzystać ze standardowej biblioteki tego języka?
- Jak programować skrypty internetowe i grafikę za jego pomocą?

Python to popularny język programowania, wykorzystywany w wielu różnych dziedzinach. Używa się go zarówno w samodzielnych programach, jak i skryptach. Jest wyposażony w dynamiczny system typów oraz automatyczne zarządzanie pamięcią. Ponadto – co istotne – jest on darmowy, łatwo przenośny i bardzo prosty w użyciu. Podobnie jak inne języki, również Python ciągle się rozwija. W tej książce przedstawiono właśnie wszystkie potrzebne i aktualne wiadomości, dotyczące wykorzystania tego języka programowania w praktyce.

Książka „Python. Wprowadzenie. Wydanie III” stanowi kompletny i szczegółowy przewodnik po podstawach języka Python. Wiedza w niej zawarta umożliwi pełne zrozumienie tego języka, co pozwoli Ci pojąć dowolne przykłady kodu na poziomie aplikacji. Zwięźle i jasno przedstawione przykłady świetnie ilustrują poszczególne kwestie i problemy, co daje solidny fundament do praktycznego zastosowania świeżo nabytych umiejętności. W niniejszym – trzecim już – wydaniu znajdziesz nowy zestaw praktycznych wskazówek i porad, dobranych w taki sposób, aby obejmowały aktualne zalecenia i najlepsze praktyki programowania. Krótko mówiąc, ta książka ma wszystko, czego potrzebujesz, aby nie tylko dokładnie poznać język Python, ale także efektywnie z niego korzystać!

- Wprowadzenie do interpretera Pythona
- Interaktywne wpisywanie kodu
- Systemowe wiersze poleceń i pliki
- Typy liczbowe Pythona
- Referencje współdzielone
- Łańcuchy znaków
- Instrukcje przypisania i instrukcje wyrażeń
- Źródła dokumentacji
- Funkcje i moduły
- Zakres i argumenty
- Programowanie zorientowane obiektowo
- Projektowanie z udziałem klas

**Zdobądź solidne podstawy języka Python i poznaj
najlepsze rozwiązania w programowaniu!**



Spis treści

Przedmowa	25
I Wprowadzenie	41
1. Pytania i odpowiedzi dotyczące Pythona	43
Dlaczego ludzie używają Pythona?	43
Jakość oprogramowania	44
Wydajność programistów	45
Czy Python jest językiem skryptowym?	45
Jakie są zatem wady Pythona?	47
Kto dzisiaj używa Pythona?	48
Co mogę zrobić za pomocą Pythona?	49
Programowanie systemowe	49
Graficzne interfejsy użytkownika	49
Skrypty internetowe	50
Integracja komponentów	50
Programowanie bazodanowe	51
Szybkie prototypowanie	51
Programowanie numeryczne i naukowe	51
Gry, grafika, AI, XML, roboty i tym podobne	51
Jakie są techniczne mocne strony Pythona?	52
Jest zorientowany obiektowo	52
Jest darmowy	52
Jest przenośny	53
Ma duże możliwości	54
Można go łączyć z innymi językami	55
Jest łatwy w użyciu	55
Jest łatwy do nauczenia się	57
Zawdzięcza swoją nazwę Monty Pythonowi	57

Jak Python wygląda na tle innych języków?	58
Podsumowanie rozdziału	59
Łamigłówka	60
Quiz	60
Odpowiedzi	60
2. Jak Python wykonuje programy?	63
Wprowadzenie do interpretera Pythona	63
Wykonywanie programu	65
Z punktu widzenia programisty	65
Z punktu widzenia Pythona	66
Kompilacja kodu bajtowego	66
Maszyna wirtualna Pythona	67
Wpływ na wydajność	67
Wpływ na proces programowania	68
Warianty modeli wykonywania	68
Alternatywne implementacje Pythona	69
CPython	69
Jython	69
IronPython	70
Narzędzia do optymalizacji wykonywania	70
Kompilator JIT Pyco	70
Translator Shedskin C++	71
Zamrożone pliki binarne	72
Przyszłe możliwości?	73
Podsumowanie rozdziału	73
Łamigłówka	74
Quiz	74
Odpowiedzi	74
3. Jak wykonuje się programy?	75
Interaktywne wpisywanie kodu	75
Wykorzystywanie sesji interaktywnej	77
Systemowe wiersze poleceń i pliki	78
Wykorzystywanie wierszy poleceń i plików	80
Skrypty wykonywalne Uniksa (#!)	81
Kliknięcie ikony pliku	83
Kliknięcie ikony w systemie Windows	83
Sztuczka z raw_input	84
Inne ograniczenia klikania ikon	86

Importowanie i przeładowywanie modułów	86
Więcej o modułach — atrybuty	88
Moduły i przestrzenie nazw	90
Uwagi na temat używania instrukcji import i reload	90
Interfejs użytkownika IDLE	91
Podstawy IDLE	91
Korzystanie z IDLE	93
Zaawansowane opcje IDLE	95
Inne IDE	95
Osadzanie wywołań	96
Zamrożone binarne pliki wykonywalne	97
Uruchamianie kodu w edytorze tekstowym	97
Inne możliwości uruchamiania	98
Przyszłe możliwości	98
Jaką opcję wybrać?	98
Podsumowanie rozdziału	99
Łamigłówka	100
Quiz	100
Odpowiedzi	100
Łamigłówka	102
Ćwiczenia do części pierwszej	102
II Typy i operacje	105
4. Wprowadzenie do typów obiektów Pythona	107
Po co korzysta się z typów wbudowanych?	108
Najważniejsze typy danych w Pythonie	109
Liczby	110
Łańcuchy znaków	111
Operacje na sekwencjach	111
Niezmienność	113
Metody specyficzne dla typu	114
Otrzymanie pomocy	115
Inne sposoby kodowania łańcuchów znaków	116
Dopasowywanie wzorców	116
Listy	117
Operacje na sekwencjach	117
Operacje specyficzne dla typu	117
Sprawdzanie granic	118
Zagnieżdżanie	119
Listy składane	119

Słowniki	120
Operacje na odwzorowaniach	120
Zagnieżdżanie raz jeszcze	121
Sortowanie kluczy — pętle for	123
Iteracja i optymalizacja	124
Brakujące klucze — testowanie za pomocą if	125
Krotki	126
Czemu służą krotki?	126
Pliki	126
Inne narzędzia podobne do plików	127
Inne typy podstawowe	128
Jak zepsuć elastyczność kodu	128
Klasy zdefiniowane przez użytkownika	129
I wszystko inne	130
Podsumowanie rozdziału	131
Łamigłówka	132
Quiz	132
Odpowiedzi	132
5. Liczby	135
Typy liczbowe Pythona	135
Literały liczbowe	136
Wbudowane narzędzia liczbowe oraz rozszerzenia	137
Operatory wyrażeń Pythona	138
Połączone operatory stosują się do priorytetów	139
Podwyrażenia grupowane są w nawiasach	139
Pomieszczone typy poddawane są konwersji	139
Przeciążanie operatorów — przegląd	140
Liczby w akcji	141
Zmienne i podstawowe wyrażenia	141
Formaty wyświetlania liczb	143
Dzielenie — klasyczne, bez reszty i prawdziwe	144
Operacje poziomego bitowego	145
Długie liczby całkowite	145
Liczby zespolone	146
Notacja szesnastkowa i ósemkowa	146
Inne wbudowane narzędzia liczbowe	147
Inne typy liczbowe	149
Liczby dziesiętne	149
Zbiory	150

Wartości Boolean	151
Dodatkowe rozszerzenia	152
Podsumowanie rozdziału	152
Łamigłówka	153
Quiz	153
Odpowiedzi	153
6. Wprowadzenie do typów dynamicznych	155
Sprawa brakujących deklaracji typu	155
Zmienne, obiekty i referencje	155
Typy powiązane są z obiektami, a nie ze zmiennymi	157
Obiekty są uwalniane	158
Referencje współdzielone	159
Referencje współdzielone a modyfikacje w miejscu	161
Referencje współdzielone a równość	162
Typy dynamiczne są wszędzie	163
Podsumowanie rozdziału	164
Łamigłówka	165
Quiz	165
Odpowiedzi	165
7. Łańcuchy znaków	167
Literały łańcuchów znaków	168
Łańcuchy znaków w apostrofach i cudzysłowach są tym samym	169
Sekwencje ucieczki reprezentują bajty specjalne	169
Surowe łańcuchy znaków blokują sekwencje ucieczki	171
Potrojne cudzysłowy i apostrofy kodują łańcuchy znaków będące wielowierszowymi blokami	172
Łańcuchy znaków Unicode pozwalają na zakodowanie większych zbiorów znaków kodowych	174
Łańcuchy znaków w akcji	176
Podstawowe operacje	176
Indeksowanie i wycinki	177
Rozszerzone wycinki — trzeci limit	179
Narzędzia do konwersji łańcuchów znaków	180
Konwersje kodu znaków	182
Modyfikowanie łańcuchów znaków	183
Formatowanie łańcuchów znaków	184
Zaawansowane formatowanie łańcuchów znaków	185
Formatowanie łańcuchów znaków w oparciu o słownik	186

Metody łańcuchów znaków	187
Przykłady metod łańcuchów znaków — modyfikowanie łańcuchów znaków	188
Przykłady metod łańcuchów znaków — analiza składniowa tekstu	190
Inne znane metody łańcuchów znaków w akcji	191
Oryginalny moduł string	192
Generalne kategorie typów	193
Typy z jednej kategorii współdzielą zbiory operacji	193
Typy zmienne można modyfikować w miejscu	194
Podsumowanie rozdziału	194
Łamigłówka	195
Quiz	195
Odpowiedzi	195
8. Listy oraz słowniki	197
Listy	197
Listy w akcji	199
Podstawowe operacje na listach	199
Indeksowanie, wycinki i macierze	200
Modyfikacja list w miejscu	201
Przypisywanie do indeksu i wycinków	201
Wywołania metod list	202
Inne popularne operacje na listach	204
Słowniki	204
Słowniki w akcji	206
Podstawowe operacje na słownikach	206
Modyfikacja słowników w miejscu	207
Inne metody słowników	208
Przykład z tabelą języków programowania	209
Uwagi na temat korzystania ze słowników	210
Wykorzystywanie słowników do symulowania elastycznych list	210
Wykorzystywanie słowników z rzadkimi strukturami danych	211
Unikanie błędów z brakującymi kluczami	211
Wykorzystywanie słowników w postaci „rekordów”	212
Inne sposoby tworzenia słowników	213
Podsumowanie rozdziału	214
Łamigłówka	215
Quiz	215
Odpowiedzi	215

9. Krotki, pliki i pozostałe	217
Krotki	217
Krotki w akcji	218
Właściwości składni krotek — przecinki i nawiasy	219
Konwersje i niezmiennosc	219
Dlaczego istnieją listy i krotki?	220
Pliki	221
Otwieranie plików	221
Wykorzystywanie plików	222
Pliki w akcji	223
Przechowywanie obiektów Pythona w plikach i przetwarzanie ich	223
Przechowywanie obiektów Pythona za pomocą pickle	225
Przechowywanie i przetwarzanie spakowanych danych binarnych w plikach	226
Inne narzędzia powiązane z plikami	227
Raz jeszcze o kategoriach typów	227
Elastyczność obiektów	228
Referencje a kopie	229
Porównania, równość i prawda	231
Znaczenie True i False w Pythonie	233
Hierarchie typów Pythona	234
Inne typy w Pythonie	235
Pułapki typów wbudowanych	236
Przypisanie tworzy referencje, nie kopie	236
Powtórzenie dodaje jeden poziom zagnieżdżenia	236
Uwaga na cykliczne struktury danych	237
Typów niezmiennych nie można modyfikować w miejscu	237
Podsumowanie rozdziału	238
Łamigłówka	239
Quiz	239
Odpowiedzi	239
Łamigłówka	240
Ćwiczenia do części drugiej	240
III Instrukcje i składnia	243
10. Wprowadzenie do instrukcji Pythona	245
Raz jeszcze o strukturze programu Pythona	245
Instrukcje Pythona	246
Historia dwóch if	246
Co dodaje Python	248

Co usuwa Python	248
Nawiasy są opcjonalne	248
Koniec wiersza jest końcem instrukcji	248
Koniec wcięcia to koniec bloku	249
Skąd bierze się składnia indentacji?	249
Kilka przypadków specjalnych	252
Przypadki specjalne dla reguły o końcu wiersza	252
Przypadki specjalne dla reguły o indentacji bloków	253
Szybki przykład — interaktywne pętle	253
Prosta pętla interaktywna	254
Wykonywanie obliczeń na danych użytkownika	255
Obsługa błędów za pomocą sprawdzania danych wejściowych	256
Obsługa błędów za pomocą instrukcji try	257
Kod zagnieżdżony na trzy poziomy głębokości	257
Podsumowanie rozdziału	258
Łamigłówka	259
Quiz	259
Odpowiedzi	259
11. Przypisanie, wyrażenia i print	261
Instrukcje przypisania	261
Formy instrukcji przypisania	262
Przypisanie sekwencji	263
Zaawansowane wzorce przypisywania sekwencji	264
Przypisanie z wieloma celami	265
Przypisanie z wieloma celami a współdzielone referencje	265
Przypisania rozszerzone	266
Przypisania rozszerzone a współdzielone referencje	268
Reguły dotyczące nazw zmiennych	268
Konwencje dotyczące nazewnictwa	270
Nazwy nie mają typu, typ mają obiekty	270
Instrukcje wyrażeń	271
Instrukcje wyrażeń i modyfikacje w miejscu	272
Instrukcje print	272
Program „Witaj, świecie!” w Pythonie	273
Przekierowanie strumienia wyjścia	274
Rozszerzenie print >> file	274
Podsumowanie rozdziału	277
Łamigłówka	278
Quiz	278
Odpowiedzi	278

12. Testy if	279
Instrukcje if	279
Ogólny format	279
Proste przykłady	280
Rozgałęzienia kodu	280
Reguły składni Pythona	282
Ograniczniki bloków	283
Ograniczniki instrukcji	284
Kilka przypadków specjalnych	285
Testy prawdziwości	285
Wyrażenie trójargumentowe if/else	287
Podsumowanie rozdziału	289
Łamigłówka	290
Quiz	290
Odpowiedzi	290
13. Pętle while i for	291
Pętle while	291
Ogólny format	291
Przykłady	292
Instrukcje break, continue, pass oraz else w pętli	293
Ogólny format pętli	293
Przykłady	293
Instrukcja pass	293
Instrukcja continue	294
Instrukcja break	295
Instrukcja else	295
Więcej o części pętli else	296
Pętle for	297
Ogólny format	297
Przykłady	298
Podstawowe zastosowanie	298
Inne typy danych	299
Przypisanie krotek w pętli for	299
Zagnieżdżone pętle for	299
Pierwsze spojrzenie na iteratory	302
Iteratory plików	302
Inne iteratory typów wbudowanych	304
Inne konteksty iteracyjne	305
Iteratory definiowane przez użytkownika	306

Techniki tworzenia pętli	306
Pętle liczników — while i range	307
Przechodzenie niewyczerpujące — range	308
Modyfikacja list — range	309
Przechodzenie równoległe — zip oraz map	310
Tworzenie słowników za pomocą funkcji zip	311
Generowanie wartości przesunięcia i elementów — enumerate	312
Listy składane — wprowadzenie	313
Podstawy list składanych	313
Wykorzystywanie list składanych w plikach	314
Rozszerzona składnia list składanych	315
Podsumowanie rozdziału	316
Łamigłówka	317
Quiz	317
Odpowiedzi	317
14. Wprowadzenie do dokumentacji	319
Źródła dokumentacji Pythona	319
Komentarze ze znakami #	320
Funkcja dir	320
Łańcuchy znaków dokumentacji — __doc__	321
Łańcuchy znaków dokumentacji zdefiniowane przez użytkownika	321
Standardy dotyczące łańcuchów znaków dokumentacji	323
Wbudowane łańcuchy znaków dokumentacji	323
PyDoc — funkcja help	324
PyDoc — raporty HTML	326
Zbiór standardowej dokumentacji	329
Zasoby internetowe	330
Publikowane książki	330
Często spotykane problemy programistyczne	330
Podsumowanie rozdziału	332
Łamigłówka	333
Quiz	333
Odpowiedzi	333
Łamigłówka	334
Ćwiczenia do części trzeciej	334

IV	Funkcje	337
15.	Podstawy funkcji	339
	Po co używa się funkcji?	340
	Tworzenie funkcji	340
	Instrukcje def	342
	Instrukcja def uruchamiana jest w czasie wykonania	342
	Pierwszy przykład — definicje i wywoływanie	343
	Definicja	343
	Wywołanie	343
	Polimorfizm w Pythonie	344
	Drugi przykład — przecinające się sekwencje	345
	Definicja	345
	Wywołania	346
	Raz jeszcze o polimorfizmie	346
	Zmienne lokalne	347
	Podsumowanie rozdziału	348
	Łamigłówka	349
	Quiz	349
	Odpowiedzi	349
16.	Zakres i argumenty	351
	Reguły dotyczące zakresu	351
	Podstawy zakresów Pythona	352
	Rozwiązywanie konfliktów w zakresie nazw — reguła LEGB	353
	Przykład zakresu	355
	Zakres wbudowany	355
	Instrukcja global	357
	Minimalizowanie stosowania zmiennych globalnych	358
	Minimalizacja modyfikacji dokonywanych pomiędzy plikami	359
	Inne metody dostępu do zmiennych globalnych	360
	Zakresy a funkcje zagnieżdżone	361
	Szczegóły dotyczące zakresów zagnieżdżonych	361
	Przykład zakresu zagnieżdżonego	361
	Funkcje fabryczne	362
	Zachowywanie stanu zakresu zawierającego	
	za pomocą argumentów domyślnych	363
	Zakresy zagnieżdżone a lambda	364
	Zakresy a domyślne wartości argumentów w zmiennych pętli	365
	Dowolne zagnieżdżanie zakresów	366

Przekazywanie argumentów	367
Argumenty a współdzielone referencje	368
Unikanie modyfikacji zmiennych argumentów	369
Symulowanie parametrów wyjścia	370
Specjalne tryby dopasowania argumentów	371
Przykłady ze słowami kluczowymi i wartościami domyślnymi	372
Słowa kluczowe	373
Wartości domyślne	373
Przykład dowolnych argumentów	374
Zbieranie argumentów	374
Rozpakowywanie argumentów	375
Łączenie słów kluczowych i wartości domyślnych	376
Przykład z funkcją obliczającą minimum	376
Pełne rozwiązanie	376
Dodatkowy bonus	378
Puenta	378
Bardziej przydatny przykład — uniwersalne funkcje działające na zbiorach	379
Dopasowywanie argumentów — szczegóły	380
Podsumowanie rozdziału	381
Łamigłówka	382
Quiz	382
Odpowiedzi	383
17. Zaawansowane zagadnienia związane z funkcjami	385
Funkcje anonimowe — lambda	385
Wyrażenia lambda	385
Po co używa się wyrażenia lambda?	387
Jak łatwo zaciemnić kod napisany w Pythonie	388
Zagnieżdżone wyrażenia lambda a zakresy	389
Zastosowanie funkcji do argumentów	390
Wbudowana funkcja apply	391
Przekazywanie argumentów ze słowami kluczowymi	391
Składnia wywołania podobna do stosowania funkcji apply	392
Odzworowywanie funkcji na sekwencje — map	392
Narzędzia programowania funkcyjnego — filter i reduce	394
Jeszcze raz listy składane — odwzorowania	395
Podstawy list składanych	395
Dodawanie testów i zagnieżdżonych pętli	396
Listy składane i macierze	398
Zrozumienie list składanych	399

Jeszcze o iteratorach — generatory	401
Przykład funkcji generatora	402
Rozszerzony protokół generatora funkcji — metoda send a next	403
Iteratory a typy wbudowane	404
Wyrażenia generatora — iteratory spotykają listy składane	404
Pomiar alternatywnych sposobów iteracji	406
Koncepcje związane z projektowaniem funkcji	408
Funkcje są obiektami — wywołania pośrednie	410
Pułapki związane z funkcjami	410
Zmienne lokalne wykrywane są w sposób statyczny	411
Wartości domyślne i obiekty zmienne	412
Funkcje bez instrukcji return	413
Zmienne pętli zakresu zawierającego	414
Podsumowanie rozdziału	414
Łamigłówka	415
Quiz	415
Odpowiedzi	415
Łamigłówka	417
Ćwiczenia do części czwartej	417
V Moduły	421
18. Moduły — wprowadzenie	423
Po co używa się modułów?	423
Architektura programu w Pythonie	424
Struktura programu	424
Importowanie i atrybuty	425
Moduły biblioteki standardowej	427
Jak działa importowanie	427
1. Odnalezienie modułu	428
Ścieżka wyszukiwania modułów	428
Lista sys.path	430
Wybór pliku modułu	430
Zaawansowane koncepcje związane z wyborem modułów	431
2. (Ewentualne) Kompilowanie	432
3. Wykonanie	432
Podsumowanie rozdziału	433
Łamigłówka	435
Quiz	435
Odpowiedzi	435

19. Podstawy tworzenia modułów	437
Tworzenie modułów	437
Użycie modułów	438
Instrukcja import	438
Instrukcja from	439
Instrukcja from *	439
Operacja importowania odbywa się tylko raz	439
Instrukcje import oraz from są przypisaniami	440
Modyfikacja zmiennych pomiędzy plikami	441
Równoważność instrukcji import oraz from	441
Potencjalne pułapki związane z użyciem instrukcji from	442
Kiedy wymagane jest stosowanie instrukcji import	443
Przestrzenie nazw modułów	443
Pliki generują przestrzenie nazw	443
Kwalifikowanie nazw atrybutów	445
Importowanie a zakresy	446
Zagnieżdżanie przestrzeni nazw	447
Przeładowywanie modułów	448
Podstawy przeładowywania modułów	449
Przykład przeładowywania z użyciem reload	450
Podsumowanie rozdziału	451
Łamigłówka	452
Quiz	452
Odpowiedzi	452
20. Pakiety modułów	453
Podstawy importowania pakietów	453
Pakiety a ustawienia ścieżki wyszukiwania	454
Pliki pakietów __init__.py	454
Przykład importowania pakietu	456
Instrukcja from a instrukcja import w importowaniu pakietów	457
Do czego służy importowanie pakietów?	458
Historia trzech systemów	458
Podsumowanie rozdziału	461
Łamigłówka	462
Quiz	462
Odpowiedzi	462

21. Zaawansowane zagadnienia związane z modułami	463
Ukrywanie danych w modułach	463
Minimalizacja niebezpieczeństw użycia from * — _X oraz __all__	464
Włączanie opcji z przyszłych wersji Pythona	464
Mieszane tryby użycia — __name__ oraz __main__	465
Testy jednostkowe z wykorzystaniem __name__	466
Modyfikacja ścieżki wyszukiwania modułów	467
Rozszerzenie import as	468
Składnia importowania względnego	468
Do czego służy importowanie względne?	469
Projektowanie modułów	471
Moduły są obiektami — metaprogramy	472
Pułapki związane z modułami	474
W kodzie najwyższego poziomu kolejność instrukcji ma znaczenie	474
Importowanie modułów za pomocą łańcucha znaków nazwy	475
Instrukcja from kopiuje nazwy, jednak łączy już nie	476
Instrukcja from * może zaciemnić znaczenie zmiennych	476
Funkcja reload może nie mieć wpływu na obiekty importowane za pomocą from	477
Funkcja reload i instrukcja from a testowanie interaktywne	478
Instrukcja reload nie jest stosowana rekurencyjnie	478
Rekurencyjne importowanie za pomocą from może nie działać	480
Podsumowanie rozdziału	481
Łamigłówka	482
Quiz	482
Odpowiedzi	482
Łamigłówka	483
Ćwiczenia do części piątej	483
VI Klasy i programowanie zorientowane obiektowo	485
22. Programowanie zorientowane obiektowo	487
Po co używa się klas?	488
Programowanie zorientowane obiektowo z dystansu	489
Wyszukiwanie dziedziczenia atrybutów	489
Klasy a instancje	492
Wywołania metod klasy	492
Tworzenie drzew klas	493
Programowanie zorientowane obiektowo	
oparte jest na ponownym wykorzystaniu kodu	495

Podsumowanie rozdziału	498
Łamigłówka	499
Quiz	499
Odpowiedzi	499
23. Podstawy tworzenia klas	501
Klasy generują większą liczbę obiektów instancji	501
Obiekty klas udostępniają zachowania domyślne	502
Obiekty instancji są rzeczywistymi elementami	502
Pierwszy przykład	503
Klasy dostosowuje się do własnych potrzeb przez dziedziczenie	505
Drugi przykład	506
Klasy są atrybutami w modułach	507
Klasy mogą przechwytywać operatory Pythona	508
Trzeci przykład	509
Po co przeciąża się operatory?	511
Najprostsza klasa Pythona na świecie	512
Podsumowanie rozdziału	514
Łamigłówka	515
Quiz	515
Odpowiedzi	515
24. Szczegóły kodu klas	517
Instrukcja class	517
Ogólna forma	517
Przykład	518
Metody	520
Przykład	521
Wywoływanie konstruktorów klas nadrzędnych	521
Inne możliwości wywoływania metod	522
Dziedziczenie	522
Tworzenie drzewa atrybutów	523
Specjalizacja odziedziczonych metod	524
Techniki interfejsów klas	524
Abstrakcyjne klasy nadrzędne	526
Przeciążanie operatorów	527
Często spotykane metody przeciążania operatorów	527
Metoda <code>__getitem__</code> przechwytuje referencje do indeksów	528
Metody <code>__getitem__</code> oraz <code>__iter__</code> implementują iterację	529

Iteratory zdefiniowane przez użytkownika	530
Wiele iteracji po jednym obiekcie	532
Metody <code>__getattr__</code> oraz <code>__setattr__</code> przechwytyją referencje do atrybutów	534
Emulowanie prywatności w atrybutach instancji	535
Metody <code>__repr__</code> oraz <code>__str__</code> zwracają reprezentacje łańcuchów znaków	536
Metoda <code>__radd__</code> obsługuje dodawanie prawostronne	537
Metoda <code>__call__</code> przechwytuje wywołania	538
Interfejsy funkcji i kod oparty na wywołaniach zwrotnych	538
Metoda <code>__del__</code> jest destruktozem	540
Przestrzenie nazw — cała historia	541
Pojedyncze nazwy — globalne, o ile nie przypisane	541
Nazwy atrybutów — przestrzenie nazw obiektów	541
Zen przestrzeni nazw Pythona — przypisania klasyfikują zmienne	542
Słowniki przestrzeni nazw	544
Łączy przestrzeni nazw	546
Bardziej realistyczny przykład	547
Podsumowanie rozdziału	550
Łamigłówka	551
Quiz	551
Odpowiedzi	551
25. Projektowanie z udziałem klas	553
Python a programowanie zorientowane obiektowo	553
Przeciążanie za pomocą sygnatur wywołań (lub nie)	554
Klasy jako rekordy	554
Programowanie zorientowane obiektowo i dziedziczenie — związek „jest”	556
Programowanie zorientowane obiektowo i kompozycja — związki „ma”	558
Raz jeszcze procesor strumienia danych	559
Programowanie zorientowane obiektowo a delegacja	562
Dziedziczenie wielokrotne	563
Klasy są obiektami — uniwersalne fabryki obiektów	566
Do czego służą fabryki?	567
Metody są obiektami — z wiązaniem i bez wiązania	568
Raz jeszcze o łańcuchach znaków dokumentacji	570
Klasy a moduły	571
Podsumowanie rozdziału	572
Łamigłówka	573
Quiz	573
Odpowiedzi	573

26. Zaawansowane zagadnienia związane z klasami	575
Rozszerzanie typów wbudowanych	575
Rozszerzanie typów za pomocą osadzania	576
Rozszerzanie typów za pomocą klas podrzędnych	576
Pseudoprywatne atrybuty klas	579
Przegląd zniekształcania nazw zmiennych	579
Po co używa się atrybutów pseudoprywatnych?	580
Klasy w nowym stylu	581
Modyfikacja wielokrotnego dziedziczenia po jednej klasie	582
Przykład wielokrotnego dziedziczenia po jednej klasie	583
Jawne rozwiązywanie konfliktów	583
Inne rozszerzenia klas w nowym stylu	585
Metody statyczne oraz metody klasy	585
Miejsca na atrybuty instancji	585
Właściwości klas	586
Nowa metoda przeciążania <code>__getattrute__</code>	588
Metody statyczne oraz metody klasy	588
Wykorzystywanie metod statycznych oraz metod klasy	590
Dekoratory funkcji	592
Przykład dekoratora	593
Pułapki związane z klasami	594
Modyfikacja atrybutów klas może mieć efekty uboczne	594
Dziedziczenie wielokrotne — kolejność ma znaczenie	595
Metody, klasy oraz zakresy zagnieżdżone	596
Przesadne opakowywanie	598
Podsumowanie rozdziału	598
Łamigłówka	599
Quiz	599
Odpowiedzi	599
Łamigłówka	600
Ćwiczenia do części szóstej	600
VII Wyjątki oraz narzędzia	607
27. Podstawy wyjątków	609
Po co używa się wyjątków?	610
Role wyjątków	610
Obsługa wyjątków w skrócie	611
Instrukcja <code>try/except/else</code>	615
Części instrukcji <code>try</code>	616
Część <code>try/else</code>	618

Przykład — zachowanie domyślne	619
Przykład — przechwytywanie wbudowanych wyjątków	620
Instrukcja try/finally	620
Przykład — działania kończące kod z użyciem try/finally	621
Połączona instrukcja try/except/finally	622
Łączenie finally oraz except za pomocą zagnieżdżenia	623
Przykład połączonego try	624
Instrukcja raise	625
Przykład — zgłaszanie i przechwytywanie wyjątków zdefiniowanych przez użytkownika	626
Przykład — przekazywanie dodatkowych danych w raise	626
Przykład — przekazywanie wyjątków za pomocą raise	627
Instrukcja assert	628
Przykład — wyłapywanie ograniczeń (ale nie błędów)	628
Menedżery kontekstu with/as	629
Podstawowe zastosowanie	629
Protokół zarządzania kontekstem	632
Podsumowanie rozdziału	633
Łamigłówka	634
Quiz	634
Odpowiedzi	634
28. Obiekty wyjątków	635
Wyjątki oparte na łańcuchach znaków	636
Wyjątki oparte na łańcuchach znaków znikają	636
Wyjątki oparte na klasach	637
Przykład wyjątku opartego na klasach	637
Po co istnieją wyjątki oparte na klasach?	639
Wbudowane klasy wyjątków	641
Określanie tekstu wyjątku	643
Przesyłanie danych oraz zachowania w instancjach	644
Przykład — dodatkowe dane w klasach i łańcuchach znaków	644
Ogólne formy instrukcji raise	646
Podsumowanie rozdziału	647
Łamigłówka	648
Quiz	648
Odpowiedzi	648

29. Projektowanie z wykorzystaniem klas	649
Zagnieżdżanie programów obsługi wyjątków	649
Przykład — zagnieżdżanie przebiegu sterowania	651
Przykład — zagnieżdżanie składniowe	651
Zastosowanie wyjątków	653
Wyjątki nie zawsze są błędami	653
Funkcje sygnalizują warunki za pomocą raise	653
Debugowanie z wykorzystaniem zewnętrznych instrukcji try	654
Testowanie kodu wewnątrz tego samego procesu	655
Więcej informacji na temat funkcji sys.exc_info	656
Wskazówki dotyczące projektowania wyjątków	656
Co powinniśmy opakować w try	656
Jak nie przechwytywać zbyt wiele — unikanie pustych except	657
Jak nie przechwytywać zbyt mało — korzystanie z kategorii opartych na klasach	659
Pułapki związane z wyjątkami	659
Wyjątki znaków dopasowywane są po tożsamości, a nie wartości	659
Przechwytywanie niewłaściwej rzeczy	661
Podsumowanie jądra języka Python	661
Zbiór narzędzi Pythona	661
Narzędzia programistyczne przeznaczone do większych projektów	663
Podsumowanie rozdziału	666
Łamigłówka	667
Quiz	667
Odpowiedzi	667
Łamigłówka	668
Ćwiczenia do części siódmej	668
Dodatki	669
A Instalacja i konfiguracja	671
B Rozwiązania ćwiczeń podsumowujących poszczególne części książki	679
Skorowidz	711

Wprowadzenie do instrukcji Pythona

Ponieważ znamy już podstawowe wbudowane typy obiektów Pythona, niniejszy rozdział rozpoczniemy od omówienia podstawowych form instrukcji tego języka. Tak jak w poprzedniej części, zaczniemy od ogólnego wprowadzenia do składni instrukcji. W kolejnych rozdziałach znajdują się bardziej szczegółowe informacje dotyczące poszczególnych instrukcji.

Mówiąc ogólnie, *instrukcje* (ang. *statement*) to coś, co piszemy w celu przekazania Pythonowi tego, co mają robić nasze programy. Jeśli program „coś robi”, to instrukcje pozwalają określić, co to konkretnie jest. Python jest językiem proceduralnym, opartym na instrukcjach. Łącząc instrukcje, określamy procedurę wykonywaną przez Pythona w celu spełnienia celów programu.

Raz jeszcze o strukturze programu Pythona

Innym sposobem zrozumienia roli instrukcji jest powrót do hierarchii wprowadzonej w rozdziale 4., który omawiał obiekty wbudowane wraz z wyrażeniami służącymi do ich przetwarzania. Niniejszy rozdział stanowi przejście o jeden poziom w górę hierarchii.

1. Programy składają się z modułów.
2. Moduły zawierają instrukcje.
3. *Instrukcje zawierają wyrażenia.*
4. Wyrażenia tworzą i przetwarzają obiekty.

Składnia Pythona składa się z instrukcji i wyrażień. Wyrażenia przetwarzają obiekty i są osadzone w instrukcjach. Instrukcje kodują większą logikę operacji programu — wykorzystują i kierują wyrażeniami do przetwarzania obiektów omawianych w poprzednich rozdziałach. Ponadto to właśnie w instrukcjach obiekty zaczynają istnieć (na przykład w wyrażeniach wewnątrz instrukcji przypisania), a niektóre instrukcje tworzą zupełnie nowe rodzaje obiektów (na przykład funkcje i klasy). Instrukcje zawsze istnieją w modułach, które z kolei same są zarządzane za pomocą instrukcji.

Instrukcje Pythona

W tabeli 10.1 zaprezentowano zbiór instrukcji Pythona¹. Niniejsza część książki omawia wpisy z tabeli od góry aż do `break` i `continue`. Niektóre z instrukcji z tej tabeli zostały już nieformalnie wprowadzone wcześniej. W tej części książki uzupełnimy pominięte szczegóły, wprowadzimy pozostałą część zbioru instrukcji proceduralnych Pythona i omówimy ogólny model składni. Instrukcje z dolnej części tabeli 10.1, dotyczące większych części programów — funkcji, klas, modułów oraz wyjątków — prowadzą do zadań programistycznych, dlatego zasługują na poświęcenie im osobnych części. Instrukcje bardziej egzotyczne, jak `exec` (kompilująca i wykonująca kod skonstruowany w postaci łańcuchów znaków), omówione są w dalszej części książki lub w dokumentacji biblioteki standardowej Pythona.

Historia dwóch `if`

Zanim zagłębimy się w szczegóły którejs z instrukcji z tabeli 10.1, chciałbym zacząć nasze omawianie składni instrukcji od pokazania, czego *nie* będziemy wpisywać do kodu Pythona, tak by można było dokonać porównania tego języka z innymi modelami składni, z jakimi można się spotkać.

Rozważmy poniższą instrukcję `if` zakodowaną w języku podobnym do C.

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

Może to być instrukcja w języku C, C++, Java, JavaScript lub Perl. Teraz przyjrzyjmy się odpowiadającej jej instrukcji z Pythona.

```
if x > y:  
    x = 1  
    y = 2
```

Pierwszą rzeczą, jaką łatwo zauważyć, jest to, że instrukcja napisana w Pythonie jest mniej, nazwijmy to, zaśmiecona — to znaczy jest w niej mniej elementów składniowych. Jest to celowe — Python jest językiem skryptowym, zatem jednym z jego celów jest ułatwienie życia programistom poprzez pisanie mniejszej ilości kodu.

Co więcej, kiedy porównamy oba modele składni, okaże się, że Python dodaje jeden dodatkowy element, a trzy elementy obecne w językach podobnych do C w Pythonie są nieobecne.

¹ Z technicznego punktu widzenia w Pythonie 2.5 `yield` z instrukcji stało się wyrażeniem, a instrukcje `try/except` i `try/finally` zostały połączone (obie były kiedyś osobnymi instrukcjami, ale teraz możemy już użyć `except` i `finally` w jednej instrukcji `try`). W Pythonie 2.6 ma zostać dodana nowa instrukcja `with/as` służąca do kodowania menedżerów kontekstu. Mówiąc ogólnie, jest to alternatywa dla operacji `try/finally` powiązanych z wyjątkami (w wersji 2.5 `with/as` jest rozszerzeniem dodatkowym i nie jest dostępne, jeśli nie włączymy go w jawny sposób, wykonując instrukcję `from __future__ import with_statement`). Więcej informacji na ten temat można znaleźć w dokumentacji Pythona. W przyszłości, w Pythonie 3.0, `print` i `exec` staną się wywołaniami funkcji, a nie instrukcjami, a nowa instrukcja `nonlocal` będzie spełniała rolę podobną do dzisiejszego `global`.

Tabela 10.1. Instrukcje Pythona

Instrukcja	Rola	Przykład
Przypisanie	Tworzenie referencji	<code>a, b, c = 'dobry', 'zły', 'paskudny'</code>
Wywołania	Wykonywanie funkcji	<code>log.write("mielonka, szynka\n")</code>
<code>print</code>	Wyświetlanie obiektów	<code>print 'The Killer', joke</code>
<code>if/elif/else</code>	Wybór działania	<code>if "python" in text: print text</code>
<code>for/else</code>	Iteracja po sekwencjach	<code>for x in mylist: print x</code>
<code>while/else</code>	Ogólne pętle	<code>while X > Y: print 'witaj'</code>
<code>pass</code>	Pusty pojemnik	<code>while True: pass</code>
<code>break, continue</code>	Przeskoki w pętli	<code>while True: if not line: break</code>
<code>try/except/finally</code>	Przechwytywanie wyjątków	<code>try: action() except: print 'Błąd w akcji'</code>
<code>raise</code>	Wywoływanie wyjątków	<code>raise endSearch, location</code>
<code>import, from</code>	Dostęp do modułów	<code>import sys from sys import stdin</code>
<code>def, return, yield</code>	Budowanie funkcji	<code>def f(a, b, c=1, *d): return a+b+c+d[0] def gen(n): for i in n, yield i*2</code>
<code>class</code>	Budowanie obiektów	<code>class subclass(Superclass): staticData = []</code>
<code>global</code>	Przestrzenie nazw	<code>def function(): global x, y x = 'nowy'</code>
<code>del</code>	Usuwanie referencji	<code>del dane[k] del dane[i:j] del obiekt.atrybut del zmienna</code>
<code>exec</code>	Wykonywanie łańcuchów znaków kodu	<code>exec "import " + modName exec code in gdict, ldict</code>
<code>assert</code>	Sprawdzanie w debugowaniu	<code>assert X > Y</code>
<code>with/as</code>	Menedżery kontekstu (Python 2.6)	<code>with open('data') as myfile: process(myfile)</code>

Co dodaje Python

Tym jednym dodatkowym elementem składniowym jest w Pythonie znak dwukropka (:). Wszystkie *instrukcje złożone* w Pythonie (czyli instrukcje z zagnieżdżonymi kolejnymi instrukcjami) pisze się zgodnie z jednym wzorcem — z nagłówkiem zakończonym dwukropkiem, po którym następuje zagnieżdżony blok kodu wcięty w stosunku do wiersza nagłówka.

```
Wiersz nagłówka:  
    Zagnieżdżony blok instrukcji
```

Dwukropek jest wymagany i pominięcie go jest chyba najczęściej popełnianym przez początkujących programistów Pythona błędem — na swoich szkoleniach i kursach widziałem go tysiące razy. Każda osoba zaczynająca swoją przygodę z Pythonem szybko zapomina o znaku dwukropka. Większość edytorów do Pythona sprawia, że błąd ten jest łatwo zauważyć, a wpisywanie dwukropka w końcu staje się nieświadomym nawykiem (do tego stopnia, że można odruchowo zacząć wpisywać dwukropki do kodu napisanego w języku C++, generując tym samym wiele zabawnych komunikatów o błędach ze strony kompilatora C++).

Co usuwa Python

Choć Python wymaga dodatkowego znaku dwukropka, istnieją trzy elementy, które muszą uwzględnić programiści języków podobnych do C, a których nie ma w Pythonie.

Nawiasy są opcjonalne

Pierwszym z nich są nawiasy wokół testów znajdujących się na górze instrukcji.

```
if (x < y)
```

Nawiasy wymagane są przez składnię wielu języków podobnych do C. W Pythonie tak nie jest — nawiasy możemy pominąć, a instrukcja nadal będzie działać.

```
if x < y
```

Z technicznego punktu widzenia, ponieważ każde wyrażenie można umieścić w nawiasach, wstawienie ich tutaj nie zaszkodzi i nie będą one potraktowane jako błąd. *Nie należy tego jednak robić* — to niepotrzebne nadużycie klawiatury, które na dodatek zdradza całemu światu, że jesteśmy byłymi programistami języka C, którzy nadal uczą się Pythona (sam takim kiedyś byłem). Sposób stosowany w Pythonie polega na całkowitym pominięciu nawiasów w tego rodzaju instrukcjach.

Koniec wiersza jest końcem instrukcji

Drugim, bardziej znaczącym elementem składni, którego nie znajdziemy w Pythonie, jest znak średnika (;). W Pythonie nie trzeba kończyć instrukcji za pomocą średników, tak jak robi się to w językach podobnych do C.

```
x = 1;
```

W Pythonie istnieje ogólna reguła mówiąca, że koniec wiersza jest automatycznie końcem instrukcji znajdującej się w tym wierszu. Innymi słowy, można opuścić średniki, a instrukcja będzie działała tak samo.

```
x = 1
```

Istnieje kilka obejść tej reguły, o czym przekonamy się za chwilę. Generalnie jednak w większości kodu w Pythonie pisze się jedną instrukcję w wierszu i średniki nie są wymagane.

Również tutaj osoby tęskniące za programowaniem w języku C (o ile to w ogóle możliwe...) mogą kontynuować używanie średników na końcu każdej instrukcji — sam język nam na to pozwala. Jednak ponownie *nie należy tego robić* (naprawdę!) — kolejny raz zdradza to, że nadal jesteśmy programistami języka C, którzy jeszcze nie przestawili się na kodowanie w Pythonie. Styl stosowany w Pythonie polega na całkowitym opuszczeniu średników.

Koniec wcięcia to koniec bloku

Trzecim i ostatnim komponentem składniowym nieobecnym w Pythonie, i chyba najbardziej niezwykłym dla byłych programistów języka C (dopóki nie poużywają go przez dziesięć minut i nie ucieszą się z jego braku), jest to, że w kodzie nie wpisuje się niczego, co jawnie oznaczałoby początek i koniec zagnieżdżonego bloku kodu. Nie musimy uwzględniać `begin/end`, `then/endif` czy umieszczać wokół kodu nawiasów klamrowych, tak jak robi się to w językach podobnych do C.

```
if (x > y) {
    x = 1;
    y = 2;
}
```

Zamiast tego w Pythonie w spójny sposób wcina się wszystkie instrukcje w danym bloku zagnieżdżonym o tę samą odległość w prawo. Python wykorzystuje fizyczne podobieństwo instrukcji do ustalenia, gdzie blok się zaczyna i gdzie się kończy.

```
if x > y:
    x = 1
    y = 2
```

Przez *indentację* rozumiemy puste białe znaki znajdujące się po lewej stronie obu zagnieżdżonych instrukcji. Pythona nie interesuje sposób indentacji (można korzystać ze spacji lub tabulatorów) ani też jej ilość (można użyć dowolnej liczby spacji lub tabulatorów). Wcięcie jednego bloku zagnieżdżonego może tak naprawdę być zupełnie inne od wcięcia innego bloku. Reguła składni mówi jedynie, że w jednym bloku zagnieżdżonym wszystkie instrukcje muszą być zagnieżdżone na tę samą odległość w prawo. Jeśli tak nie jest, otrzymamy błąd składni i kod nie zostanie wykonany, dopóki nie naprawimy indentacji w spójny sposób.

Skąd bierze się składnia indentacji?

Reguła indentacji może programistom języków podobnych do C na pierwszy rzut oka wydać się niezwykła, jednak jest ona celową cechą Pythona oraz jednym ze sposobów, w jaki Python wymusza na programistach tworzenie jednolitego, regularnego i czytelnego kodu. Oznacza to, że kod musi być wyrównany w pionie, w kolumnach, zgodnie ze swoją strukturą logiczną. Rezultat jest taki, że kod staje się bardziej spójny i czytelny (w przeciwieństwie do kodu napisanego w językach podobnych do C).

Mówiąc inaczej, wyrównanie kodu zgodnie z jego strukturą logiczną jest podstawowym narzędziem uczynienia go czytelnym i tym samym łatwym w ponownym użyciu oraz późniejszym utrzymywaniu — zarówno przez nas samych, jak i przez inne osoby. Nawet osoby,

które po skończeniu lektury niniejszej książki nigdy nie będą używać Pythona, powinny nabrać nawyku wyrównywania kodu w celu zwiększenia jego czytelności w dowolnym języku o strukturze blokowej. Python wymusza to, gdyż jest to częścią jego składni, jednak ma to znaczenie w każdym języku programowania i ogromny wpływ na użyteczność naszego kodu.

Doświadczenia każdej osoby mogą być różne, jednak kiedy ja byłem pełnoetatowym programistą, byłem zatrudniony przede wszystkim przy pracy z dużymi, starymi programami w języku C++, tworzonymi przez długie lata przez wiele różnych osób. Co było nie do uniknięcia, prawie każdy programista miał swój własny styl indentacji kodu. Często proszono mnie na przykład o zmianę pętli `while` zakodowanej w języku C, która rozpoczynała się w następujący sposób:

```
while (x > 0) {
```

Zanim jeszcze przejdziemy do samej indentacji, powiem, że istnieją trzy czy cztery sposoby układania nawiasów klamrowych w językach podobnych do C. Wiele organizacji prowadzi niemal polityczne debaty i tworzy podręczniki opisujące standardy, które powinny sobie z tym radzić (co wydaje się nieco przesadne, zważając na to, że trudno uznać to za problem programistyczny). Ignorując te spory, przejdźmy do scenariusza, z jakim często spotykałem się w kodzie napisanym w języku C++. Pierwsza osoba pracująca nad tym kodem wciñała pętlę o cztery spacje.

```
while (x > 0) {
    -----;
    -----;
```

Ta osoba w końcu awansowała, zajęła się zarządzaniem, a jej miejsce zajął ktoś, kto wolał wcięcia jeszcze bardziej przesunięte w prawo.

```
while (x > 0) {
    -----;
    -----;
        -----;
        -----;
```

I ta osoba w pewnym momencie zmieniła pracę, a jej zadania przejął ktoś, kto wolał mniejsze wcięcia.

```
while (x > 0) {
    -----;
    -----;
        -----;
        -----;
    -----;
    -----;
}
```

I tak w nieskończoność. Blok ten kończy się nawiasem klamrowym (`}`), co oczywiście sprawia, że jest on kodem ustrukturyzowanym blokowo (przynajmniej teoretycznie). W każdym języku ustrukturyzowanym blokowo (w Pythonie i innych), jeśli zagnieżdżone bloki nie są wcięte w spójny sposób, stają się trudne do interpretacji, modyfikacji czy ponownego użycia. Czytelność ma duże znaczenie i indentacja jest jej istotnym komponentem.

Poniżej znajduje się kolejny przykład, który mógł nas zabołec w przeszłości, jeśli programowaliśmy kiedyś w języku podobnym do C. Rozważmy poniższą instrukcję języka C:

```
if (x)
    if (y)
        instrukcja1;
    else
        instrukcja2;
```

Z którym `if` powiązane jest `else`? Co może być zaskoczeniem, `else` jest dopełnieniem zagnieżdżonej instrukcji `if (y)`, choć wizualnie wydaje się przynależec do zewnętrznej instrukcji `if (x)`. To klasyczna pułapka języka C, która może prowadzić do całkowicie niepoprawnej interpretacji kodu przez czytelnika i jego modyfikacji w niepoprawny sposób, co może nie zostać wykryte do momentu, gdy marsjański łazik rozbije się na wielkiej skale!

Takie coś nie może się jednak zdarzyć w Pythonie. Ponieważ indentacja jest znacząca, to, jak wygląda kod, przedstawia to, jak działa. Rozważmy odpowiednik powyższego kodu w Pythonie:

```
if x:
    if y:
        instrukcja1
    else:
        instrukcja2
```

W tym przykładzie `if`, z którym `else` wyrównane jest w pionie, to `if`, z którym `else` jest powiązane logicznie (jest to zewnętrzne `if x`). W pewnym sensie Python jest językiem typu WYSIWYG — to, co widzimy, jest tym, co otrzymujemy, ponieważ kod wykonywany jest tak, jak wygląda, bez względu na jego autora.

Jeśli te argumenty nie były w stanie przekonać kogoś o wyższości składni Pythona, podam jeszcze jedną anegdotę. Na początku mojej kariery zawodowej pracowałem w firmie rozwijającej oprogramowanie w języku C, w którym spójna indentacja nie jest wymagana. Mimo to, kiedy pod koniec dnia przesyłaliśmy kod do systemu kontroli wersji, wykorzystywano zautomatyzowany skrypt analizujący indentację w kodzie. Jeśli skrypt zauważył, że nie wcinamy kodu w sposób spójny, następnego dnia czekał na nas e-mail w tej sprawie, który trafiał również do naszych szefów.

Dlaczego o tym piszę? Nawet jeśli język programowania tego nie wymaga, dobrzy programiści wiedzą, że spójna indentacja kodu ma ogromny wpływ na jego czytelność i jakość. To, że Python przenosi tę kwestię na poziom składni, przez większość osób uznawane jest za wielką zaletę.

Wreszcie należy pamiętać, że prawie każdy używany obecnie edytor tekstu dla programistów ma wbudowaną obsługę modelu składni Pythona. W IDLE wiersze kodu są wcinane automatycznie, kiedy piszemy blok zagnieżdżony. Naciśnięcie przycisku *Backspace* powraca o jeden poziom wcięcia i można również ustawić, jak daleko do prawej strony IDLE wcina instrukcje zagnieżdżonego bloku.

Nie ma bezwzględnego standardu określającego sposób wcinania kodu. Często stosuje się cztery spacje lub jeden tabulator na poziom, jednak to każdy z nas decyduje, w jaki sposób i na jaką odległość wcinać kod. Dla bloków bardziej zagnieżdżonych odległość ta może być większa, dla bloków bliższych zewnętrznemu — mniejsza. Co więcej, wstawianie tabulatorów zamiast nawiasów klamrowych nie jest w praktyce trudniejsze dla narzędzi zwracających kod w Pythonie. Generalnie wystarczy robić to samo co w językach podobnych do C — należy się tylko pozbyć nawiasów klamrowych, a kod będzie spełniał reguły składni Pythona.

Kilka przypadków specjalnych

Jak wspomniano wcześniej, w modelu składni Pythona:

- koniec wiersza kończy instrukcję znajdującą się w tym wierszu (bez konieczności użycia średników),
- instrukcje zagnieżdżone są łączone w bloki i wiązane ze sobą za pomocą fizycznego wcięcia (bez konieczności użycia nawiasów klamrowych).

Te reguły decydują o prawie całym kodzie napisanym w Pythonie, z jakim można się spotkać w praktyce. Python udostępnia również kilka reguł specjalnego przeznaczenia, które pozwalają na dostosowanie instrukcji i zagnieżdżonych bloków instrukcji do własnych potrzeb.

Przypadki specjalne dla reguły o końcu wiersza

Choć instrukcje normalnie pojawiają się po jednej na wiersz, można również umieścić w wierszu kodu Pythona więcej niż jedną instrukcję, rozdzielając je od siebie średnikami.

```
a = 1; b = 2; print a + b # Trzy instrukcje w wierszu
```

To jedyne miejsce, w którym w Pythonie wymagane są średniki — jako *separatory instrukcji*. Działa to jednak tylko wtedy, gdy połączone w ten sposób instrukcje nie są instrukcjami złożonymi. Innymi słowy, można połączyć ze sobą jedynie proste instrukcje, takie jak przypisanie, wyświetlanie za pomocą `print` czy wywołania funkcji. Instrukcje złożone nadal muszą pojawiać się w osobnych wierszach (w przeciwnym razie w jednym wierszu można by było umieścić cały program, co nie przysporzyłoby nam popularności wśród współpracowników).

Druga reguła specjalna dla instrukcji jest odwrotna: jedna instrukcja może rozciągać się na kilka wierszy. By to zadziałało, wystarczy umieścić część instrukcji w parze nawiasów — zwykłych (`()`), kwadratowych (`[]`) lub klamrowych (`{}`). Kod umieszczony w tych konstrukcjach może znajdować się w kilku wierszach. Instrukcja nie kończy się, dopóki Python nie dojdzie do wiersza zawierającego zamykającą część nawiasu. Przykładem może być rozciągający się na kilka wierszy literał listy.

```
mlist = [111,  
         222,  
         333]
```

Ponieważ kod umieszczony jest w parze nawiasów kwadratowych, Python po prostu przechodzi do kolejnego wiersza aż do momentu napotkania nawiasu zamykającego. W ten sposób na kilka wierszy mogą się rozciągać również słowniki, a zwykłe nawiasy mogą mieścić krotki, wywołania funkcji i wyrażenia. Indentacja wiersza z kontynuacją nie ma znaczenia, choć zdrowy rozsądek nakazuje jakoś wyrównać ze sobą kolejne wiersze dla celów czytelności.

Nawiasy są wszechstronnym narzędziem. Ponieważ można w nich umieścić dowolne wyrażenie, samo wstawienie lewego nawiasu pozwala na przejście do kolejnego wiersza i kontynuowanie instrukcji tam.

```
X = (A + B +  
     C + D)
```

Ta technika działa zresztą również w przypadku instrukcji złożonych. Kiedy tylko potrzebujemy zakodować duże wyrażenie, wystarczy opakować je w nawiasy, by móc je kontynuować w kolejnym wierszu.

```
if (A == 1 and
    B == 2 and
    C == 3):
    print 'mielonka' * 3
```

Starsza reguła pozwala również na kontynuację w następnym wierszu, kiedy poprzedni kończy się ukośnikiem lewym.

```
X = A + B + \  
    C + D
```

Ta technika alternatywna jest przestarzała i raczej już nie lubiana, ponieważ trudno jest zauważyć i utrzymywać ukośniki lewe, a do tego dość bezwzględna (po ukośniku nie może być spacji). Jest to również kolejny powrót do języka C, gdzie jest ona często wykorzystywana w makrach „#define”. W świecie Pythona należy zachowywać się jak programista Pythona, a nie języka C.

Przypadki specjalne dla reguły o indentacji bloków

Jak wspomniano wcześniej, instrukcje w zagnieżdżonym bloku kodu są zazwyczaj wiązane ze sobą dzięki wcinaniu na tę samą odległość w prawą stronę. W specjalnym przypadku ciało instrukcji złożonej może pojawić się w tym samym wierszu co jej nagłówek, po znaku dwukropka.

```
if x > y: print x
```

Pozwala to na kodowanie jednowierszowych instrukcji `if` czy pętli. Tutaj jednak zadziała to tylko wtedy, gdy ciało instrukcji złożonej nie zawiera żadnych instrukcji złożonych. Mogą się tam znajdować jedynie proste instrukcje — przypisania, instrukcje `print`, wywołania funkcji i tym podobne. Większe instrukcje nadal muszą być umieszczane w osobnych wierszach. Dodatkowe części instrukcji złożonych (na przykład część `else` z `if`, z którą spotkamy się później) również muszą znajdować się w osobnych wierszach. Ciało instrukcji może składać się z kilku prostych instrukcji rozdzielonych średnikami, jednak zazwyczaj nie jest to pochwalone.

Ogólnie rzecz biorąc, nawet jeśli nie zawsze jest to wymagane, jeśli będziemy umieszczać każdą instrukcję w osobnym wierszu i zawsze wcinąć zagnieżdżone bloki, nasz kod będzie łatwiejszy do odczytania i późniejszej modyfikacji. By zobaczyć najważniejszy i najczęściej spotykany wyjątek od jednej z tych reguł (użycie jednowierszowej instrukcji `if` w celu wyjścia z pętli), przejdźmy do kolejnego podrozdziału i zajmijmy się pisaniem prawdziwego kodu.

Szybki przykład — interaktywne pętle

Wszystkie te reguły składni zobaczymy w działaniu, kiedy w kolejnych rozdziałach będziemy omawiać określone instrukcje złożone Pythona. Działają one w ten sam sposób w całym języku. Na początek zajmiemy się krótkim, realistycznym przykładem demonstrującym sposób łączenia składni i zagnieżdżania instrukcji, a także wprowadzającym przy okazji kilka instrukcji.

Prosta pętla interaktywna

Przypuśćmy, że poproszono nas o napisanie w Pythonie programu wchodzącego w interakcję z użytkownikiem w oknie konsoli. Być może będziemy przyjmować dane wejściowe w celu przesłania ich do bazy danych bądź odczytywać liczby wykorzystane w obliczeniach. Bez względu na cel potrzebna nam będzie pętla odczytująca dane wejściowe wpisywane przez użytkownika na klawiaturze i wyświetlająca dla nich wynik. Innymi słowy, musimy utworzyć klasyczną pętlę odczytaj-oblicz-wyświetl.

W Pythonie typowy kod takiej pętli interaktywnej może wyglądać jak poniższy przykład.

```
while True:
    reply = raw_input('Wpisz tekst:')
    if reply == 'stop': break
    print reply.upper( )
```

Kod ten wykorzystuje kilka nowych koncepcji.

- W kodzie użyto pętli `while` — najbardziej ogólnej instrukcji pętli Pythona. Instrukcję `while` omówimy bardziej szczegółowo później. Mówiąc w skrócie, zaczyna się ona od słowa `while`, a po nim następuje wyrażenie, którego wynik interpretowany jest jako prawda lub fałsz. Później znajduje się zagnieżdżony blok kodu powtarzany, dopóki test znajdujący się na górze jest prawdą (słowo `True` z przykładu jest zawsze prawdą).
- Wbudowana funkcja `raw_input`, z którą spotkaliśmy się już wcześniej, wykorzystana została tutaj do wygenerowania danych wejściowych z konsoli. Wyświetla ona w charakterze zachęty łańcuch znaków będący opcjonalnym argumentem i zwraca odpowiedź wpisaną przez użytkownika w postaci łańcucha znaków.
- W kodzie pojawia się również jednowierszowa instrukcja `if` wykorzystująca regułę specjalną dotyczącą zagnieżdżonych bloków. Ciało instrukcji `if` pojawia się po dwukropku w jej nagłówku, zamiast znajdować się w kolejnym, wciętych wierszu. Oba alternatywne sposoby zadziałają, jednak dzięki metodzie zastosowanej powyżej zaoszczędziliśmy jeden wiersz.
- Instrukcja `break` Pythona wykorzystywana jest do natychmiastowego wyjścia z pętli. Powoduje ona całkowite wyskoczenie z instrukcji pętli i program kontynuowany jest po pętli. Bez takiej instrukcji wyjścia pętla byłaby nieskończona, ponieważ wynik jej testu będzie zawsze prawdziwy.

W rezultacie takie połączenie instrukcji oznacza: „Wczytaj wiersze wpisane przez użytkownika i wyświetl je zapisane wielkimi literami, dopóki nie wpisze on słowa `stop`”. Istnieją inne sposoby zakodowania takiej pętli, jednak metoda zastosowana powyżej jest w Pythonie często spotykana.

Warto zauważyć, że wszystkie trzy wiersze zagnieżdżone pod wierszem nagłówka instrukcji `while` są wcinane na tę samą odległość. Ponieważ są one wyrównane w pionie jak jedna kolumna, są blokiem kodu powiązanego z testem `while` i powtarzanego. Blok ciała pętli zostaje zakończony albo przez koniec pliku źródłowego, albo przez umieszczenie mniej wciętej instrukcji.

Po wykonaniu kodu możemy otrzymać interakcję podobną do poniższej.

```
Wpisz tekst:mielonka
MIELONKA
Wpisz tekst:42
42
Wpisz tekst:stop
```

Wykonywanie obliczeń na danych użytkownika

Nasz skrypt działa, jednak teraz założmy, że zamiast zmiany tekstowego łańcucha znaków na zapisany wielkimi literami wolelibyśmy wykonać jakieś obliczenia na danych liczbowych — na przykład podnosząc je do kwadratu. By osiągnąć zamierzony efekt, możemy spróbować z poniższymi instrukcjami.

```
>>> reply = '20'
>>> reply ** 2
...pominięto tekst błędu...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Takie coś nie zadziała w naszym skrypcie, ponieważ Python nie przekształci typów obiektów w wyrażeniach, jeśli wszystkie one nie są typami liczbowymi — a tak nie jest, ponieważ dane wpisywane przez użytkownika są zawsze w skrypcie zwracane jako łańcuchy znaków. Nie możemy podnieść łańcucha cyfr do potęgi, dopóki ręcznie nie przekształcimy go na liczbę całkowitą.

```
>>> int(reply) ** 2
400
```

Mając takie informacje, możemy teraz poprawić naszą pętlę w taki sposób, by wykonywała ona niezbędne obliczenia.

```
while True:
    reply = raw_input('Wpisz tekst:')
    if reply == 'stop': break
    print int(reply) ** 2
print 'Koniec'
```

Ten skrypt wykorzystuje jednowierszową instrukcję `if` do wyjścia z pętli w momencie wpisania słowa „stop”, jednak przy okazji konwertuje również dane wejściowe na postać liczbową w celu umożliwienia obliczeń. Wersja ta dodaje także komunikat końcowy umieszczony na dole. Ponieważ instrukcja `print` z ostatniego wiersza nie jest wcięta na tę samą odległość co zagnieżdżony blok kodu, nie jest ona uważana za część ciała pętli i zostanie wykonana tylko raz — po wyjściu z pętli.

```
Wpisz tekst:2
4
Wpisz tekst:40
1600
Wpisz tekst:stop
Koniec
```


Obsługa błędów za pomocą sprawdzania danych wejściowych

Jak na razie wszystko działa, ale co się stanie, kiedy dane wejściowe będą niepoprawne?

```
Wpisz tekst:xxx
...pominięto tekst błędu...
ValueError: invalid literal for int() with base 10: 'xxx'
```

Wbudowana funkcja `int` w momencie wystąpienia błędu zwraca tutaj wyjątek. Jeśli chcemy, by nasz skrypt miał większe możliwości, możemy z wyprzedzeniem sprawdzić zawartość łańcucha znaków za pomocą metody obiektu łańcucha znaków o nazwie `isdigit`.

```
>>> S = '123'
>>> T = 'xxx'
>>> S.isdigit( ), T.isdigit( )
(True, False)
```

Daje nam to również pretekst do dalszego zagnieżdżenia instrukcji w naszym przykładzie. Poniższa nowa wersja naszego interaktywnego skryptu wykorzystuje pełną instrukcję `if` do obejścia problemu wyjątków pojawiających się w momencie wystąpienia błędu.

```
while True:
    reply = raw_input('Wpisz tekst:')
    if reply == 'stop':
        break
    elif not reply.isdigit( ):
        print 'Niepoprawnie!' * 5
    else:
        print int(reply) ** 2
print 'Koniec'
```

Instrukcję `if` przestudiujemy szczegółowo w rozdziale 12. Jest ona dość łatwym narzędziem służącym do kodowania logiki w skryptach. W pełnej formie składa się ze słowa `if`, po którym następuje test, powiązany blok kodu, jeden lub większa liczba opcjonalnych testów `elif` (od „else if”) i bloków kodu, a na dole opcjonalna część `else` z powiązaniem blokiem kodu, który służy za wynik domyślny. Kiedy pierwszy test zwraca wynik będący prawdą, Python wykonuje blok kodu z nim powiązany — od góry do dołu. Jeśli wszystkie testy będą zwracały wyniki będące fałszem, wykonywany jest kod z części `else`.

Części `if`, `elif` i `else` w powyższym przykładzie są powiązanymi częściami tej samej instrukcji, ponieważ wszystkie są ze sobą wyrównane w pionie (to znaczy mają ten sam poziom wcięcia). Instrukcja `if` rozciąga się od słowa `if` do początku instrukcji `print` w ostatnim wierszu skryptu. Z kolei cały blok `if` jest częścią pętli `while`, ponieważ w całości wcięty jest pod wierszem nagłówka tej pętli. Zagnieżdżanie instrukcji z czasem stanie się dla każdego naturalne.

Kiedy wykonamy nasz nowy skrypt, jego kod przechwyci błąd przed jego wystąpieniem i wyświetli (dość głupi) komunikat w celu podkreślenia tego.

```
Wpisz tekst:5
25
Wpisz tekst:xyz
Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!
Wpisz tekst:10
100
Wpisz tekst:stop
```

Obsługa błędów za pomocą instrukcji try

Powyższe rozwiązanie działa, jednak, jak zobaczymy w dalszej części książki, najbardziej uniwersalnym sposobem obsługi wyjątków w Pythonie jest przechwytywanie ich i poradzenie sobie z błędem za pomocą instrukcji `try`. Instrukcję tę omówimy bardziej dogłębnie w ostatniej części książki, jednak już teraz możemy pokazać, że użycie tutaj `try` może sprawić, iż kod niektórym osobom wyda się prostszy od poprzedniej wersji.

```
while True:
    reply = raw_input('Wpisz tekst:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print 'Niepoprawnie!' * 5
    else:
        print int(reply) ** 2
print 'Koniec'
```

Ta wersja działa dokładnie tak samo jak poprzednia, jednak zastąpiliśmy dosłowne sprawdzanie błędu kodem zakładającym, że konwersja będzie działała, i opakowaliśmy go kodem z obsługą wyjątku, który zatroszczy się o przypadki, kiedy konwersja nie zadziała. Instrukcja `try` składa się ze słowa `try`, następującego po nim głównego bloku kodu (z działaniem, jakie próbujemy uzyskać), później z części `except` podającej kod obsługujący błędy i części `else`, która jest wykonywana, kiedy żaden wyjątek nie zostanie zgłoszony w części `try`. Python najpierw próbuje wykonać część `try`, a później albo część `except` (jeśli wystąpi wyjątek), albo `else` (jeśli wyjątek się nie pojawi).

Jeśli chodzi o zagnieżdżanie instrukcji, ponieważ poziom wcięcia `try`, `except` i `else` jest taki sam, wszystkie one uznawane są za część tej samej instrukcji `try`. Warto zauważyć, że część `else` powiązana jest tutaj z `try`, a nie z `if`. Jak zobaczymy później, `else` może się w Pythonie pojawiać w instrukcjach `if`, ale także w instrukcjach `try` i pętlach — to indentacja informuje nas, której instrukcji jest częścią.

Do instrukcji `try` powrócimy w dalszej części książki. Na razie warto być świadomym tego, że ponieważ `try` można wykorzystać do przechwycenia dowolnego błędu, instrukcja ta redukuje ilość kodu sprawdzającego błędy, jaki musimy napisać. Jest także bardzo uniwersalnym sposobem radzenia sobie z niezwykłymi przypadkami.

Kod zagnieżdżony na trzy poziomy głębokości

Przyjrzyjmy się teraz ostatniej mutacji naszego skryptu. Zagnieżdżanie może być jeszcze głębsze — możemy na przykład rozgałęzić jedną z alternatyw w oparciu o względną wielkość poprawnych danych wejściowych.

```
while True:
    reply = raw_input('Wpisz tekst:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print 'Niepoprawnie!' * 5
    else:
        num = int(reply)
        if num < 20:
            print 'mało'
        else:
            print num ** 2
print 'Koniec'
```

Ta wersja zawiera instrukcję `if` zagnieżdżoną w części `else` innej instrukcji `if`, która z kolei jest zagnieżdżona w pętli `while`. Kiedy kod jest warunkowy lub powtarzany, tak jak ten, po prostu wcinamy go jeszcze dalej w prawo. W rezultacie otrzymujemy coś podobnego do poprzedniej wersji, ale dla liczb mniejszych od 20 wyświetlony zostanie komunikat „mało”.

```
Wpisz tekst:19
mało
Wpisz tekst:20
400
Wpisz tekst:mieLonka
Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!
Wpisz tekst:stop
Koniec
```

Podsumowanie rozdziału

Powyższe informacje kończą nasze krótkie omówienie podstaw składni instrukcji Pythona. W niniejszym rozdziale wprowadziliśmy ogólne reguły kodowania instrukcji oraz bloków kodu. Jak widzieliśmy, w Pythonie zazwyczaj umieszcza się jedną instrukcję na wiersz i wcina wszystkie instrukcje zagnieżdżonego bloku kodu na tę samą odległość (indentacja jest częścią składni Pythona). Przyjrzelśmy się również kilku odstępstwom od tych reguł, w tym wierszom z kontynuacją oraz jednowierszowym testom i pętlom. Wreszcie zastosowaliśmy te koncepcje w praktyce w interaktywnym skrypcie, który zademonstrował kilka instrukcji i pokazał nam, jak tak naprawdę działa składnia instrukcji.

W kolejnym rozdziale zaczniemy zagłębiać się w instrukcje, omawiając bardziej szczegółowo każdą z podstawowych instrukcji proceduralnych Pythona. Jak już jednak widzieliśmy, wszystkie instrukcje zachowują się zgodnie z ogólnymi regułami zaprezentowanymi tutaj.

Quiz

1. Jakie trzy elementy wymagane są w językach podobnych do C, ale pomijane w Pythonie?
2. W jaki sposób w Pythonie normalnie kończy się instrukcje?
3. W jaki sposób instrukcje z zagnieżdżonego bloku kodu są zazwyczaj ze sobą wiązane w Pythonie?
4. W jaki sposób możemy rozciągnąć instrukcję na kilka wierszy?
5. W jaki sposób można utworzyć instrukcję złożoną zajmującą jeden wiersz?
6. Czy istnieje jakiś powód uzasadniający umieszczenie średnika na końcu instrukcji w Pythonie?
7. Do czego służy instrukcja `try`?
8. Co jest najczęściej popełnianym przez osoby początkujące błędem w kodowaniu w Pythonie?

Odpowiedzi

1. Języki podobne do C wymagają stosowania nawiasów wokół testów w niektórych instrukcjach, średników na końcu instrukcji i nawiasów klamrowych wokół zagnieżdżonych bloków kodu.
2. Koniec wiersza kończy instrukcję umieszczoną w tym wierszu. Alternatywnie, jeśli w tym samym wierszu znajduje się większa liczba instrukcji, można je zakończyć średnikami. W podobny sposób, kiedy instrukcja rozciąga się na wiele wierszy, trzeba ją zakończyć za pomocą zamknięcia pary nawiasów.
3. Instrukcje w bloku zagnieżdżonym są wcinane o taką samą liczbę tabulatorów lub spacji.
4. Instrukcja może rozciągać się na kilka wierszy dzięki umieszczeniu jej części w nawiasach zwykłych, kwadratowych lub klamrowych. Instrukcja taka kończy się, kiedy Python napotka wiersz zawierający zamykającą część pary nawiasów.
5. Ciało instrukcji złożonej można przesunąć do wiersza nagłówka po dwukropku, jednak tylko wtedy, gdy nie zawiera ono żadnych instrukcji złożonych.
6. Możemy to zrobić tylko wtedy, gdy chcemy zmieścić w jednym wierszu kodu więcej niż jedną instrukcję. Ale nawet wówczas działa to tylko w sytuacji, w której żadna z instrukcji nie jest złożona, i jest odradzane, ponieważ prowadzi do kodu trudniejszego w odczycie.
7. Instrukcja `try` wykorzystywana jest do przechwytywania wyjątków (błędów) i radzenia sobie z nimi w skrypcie Pythona. Zazwyczaj jest alternatywą dla ręcznego sprawdzania błędów w kodzie.
8. Błędem najczęściej popełnianym przez osoby początkujące jest zapomnienie o dodaniu dwukropka na końcu wiersza nagłówka instrukcji złożonej.