

Spis treści

O autorze	7
O korektorze merytorycznym	8
Podziękowania	9
Wprowadzenie	11
Rozdział 1 Kodowanie pythoniczne	13
Tworzenie pythonicznego kodu	13
Nazewnictwo	14
Wyrażenia i instrukcje	16
Pythoniczny styl kodowania	19
Komentarze dokumentacyjne	24
Komentarze dokumentacyjne do modułów	26
Komentarze dokumentacyjne do klas	26
Komentarze dokumentacyjne do funkcji	27
Przydatne narzędzia dokumentacyjne	28
Pythoniczne struktury sterujące	28
Wyrażenia listowe	28
Nie twórz skomplikowanych wyrażeń listowych	29
Kiedy stosować wyrażenia lambda?	31
Kiedy stosować generatory, a kiedy wyrażenia listowe?	31
Dlaczego nie należy stosować instrukcji else w pętlach?	32
Dlaczego warto stosować funkcję range() w języku Python 3?	34
Zgłaszanie wyjątków	35
Często zgłaszane wyjątki	36
Obsługuj wyjątki za pomocą instrukcji finally	37
Twórz własne klasy wyjątków	37
Obsługuj konkretne wyjątki	39

	Zwracaj uwagę na zewnętrzne wyjątki	40
	Twórz jak najmniejsze bloki try	41
	Podsumowanie	42
Rozdział 2	Struktury danych	43
	Popularne struktury danych	43
	Zbiory i wydajny kod	43
	Przetwarzanie danych za pomocą struktury namedtuple	45
	Typ str i znaki diakrytyczne	47
	Zamiast list stosuj iteratory	48
	Przetwarzaj listy za pomocą funkcji zip()	50
	Wykorzystuj zalety wbudowanych funkcji	51
	Zalety słownika	53
	Kiedy używać słownika zamiast innych struktur?	53
	Kolekcje	53
	Słowniki uporządkowany, domyślny i zwykły	56
	Słownik jako odpowiednik instrukcji switch	57
	Scalanie słowników	58
	Czytelne wyświetlanie zawartości słownika	59
	Podsumowanie	60
Rozdział 3	Jak pisać lepsze funkcje i klasy?	61
	Funkcje	61
	Twórz małe funkcje	61
	Twórz generatory	63
	Używaj wyjątku zamiast wyniku None	64
	Stosuj w argumentach klucze i wartości domyślne	65
	Nie twórz funkcji jawnie zwracających wynik None	66
	Krytycznie podchodź do tworzonych funkcji	68
	Stosuj w wyrażeniach funkcje lambda	70
	Klasy	71
	Jak duża powinna być klasa?	71
	Struktura klasy	72
	Właściwe użycie dekoratora @property	74
	Kiedy należy stosować metody statyczne?	75
	Dziedziczenie klas abstrakcyjnych	76
	Odwołania do stanu klasy przy użyciu dekoratora @classmethod	77
	Atrybuty publiczne zamiast prywatnych	78
	Podsumowanie	79
Rozdział 4	Praca z modułami i metaklasami	81
	Moduły i metaklasy	81
	Porządkowanie kodu za pomocą modułów	82
	Zalety pliku __init__.py	84

Importowanie funkcji i klas z modułów	86
Blokowanie importu całego modułu za pomocą metaklasy <code>__all__</code>	87
Kiedy stosować metaklasy?	88
Weryfikowanie podklas za pomocą metody <code>__new__()</code>	89
Dlaczego atrybut <code>__slots__</code> jest tak przydatny?	91
Modyfikowanie funkcjonowania klasy za pomocą metaklasy	93
Deskryptory w języku Python	94
Podsumowanie	96

Rozdział 5 Dekoratory i menedżery kontekstu 97

Dekoratory	97
Czym są dekoratory i dlaczego są tak przydatne?	98
Korzystanie z dekoratorów	99
Modyfikowanie działania funkcji za pomocą dekoratorów	101
Stosowanie kilku dekoratorów jednocześnie	102
Dekorowanie funkcji z argumentami	103
Używaj dekoratorów z biblioteki	104
Dekoratory obsługujące stan klasy i weryfikujące poprawność danych	106
Menedżery kontekstu	108
Zalety menedżerów kontekstu	108
Tworzenie menedżera kontekstu od podstaw	109
Tworzenie menedżera kontekstu za pomocą biblioteki <code>contextlib</code>	111
Praktyczne przykłady użycia menedżera kontekstu	111
Podsumowanie	114

Rozdział 6 Generatory i iteratory 115

Zalety generatorów i iteratorów	115
Iteratory	115
Generatory	117
Kiedy stosować iteratory?	118
Moduł <code>itertools</code>	119
Dlaczego generatory są tak przydatne?	121
Wyrażenia listowe i iteratory	122
Zalety instrukcji <code>yield</code>	122
Instrukcja <code>yield from</code>	123
Instrukcja <code>yield</code> jest szybka	123
Podsumowanie	124

Rozdział 7 Nowe funkcjonalności języka Python 125

Programowanie asynchroniczne	125
Wprowadzenie do programowania asynchronicznego	126
Jak to działa?	128
Obiekty oczekiwalne	133
Biblioteki do tworzenia kodu asynchronicznego	139

Python i typy danych	143
Typy danych w Pythonie	143
Moduł typing	144
Czy typy danych spowalniają kod?	145
Jak dzięki modułowi typing można pisać lepszy kod?	146
Metoda super()	147
Lepsza obsługa ścieżek dzięki bibliotece pathlib	147
print() jest teraz funkcją	147
f-ciągi	147
Obowiązkowe argumenty pozycyjne	148
Kontrolowana kolejność elementów w słownikach	148
Iteracyjne rozpakowywanie struktur	149
Podsumowanie	149
Rozdział 8	
Diagnostyka i testy kodu	151
Diagnostyka	151
Narzędzia diagnostyczne	152
Funkcja breakpoint()	155
Moduł logging zamiast funkcji print()	155
Identyfikowanie słabych punktów kodu za pomocą metryk	159
Do czego przydaje się środowisko IPython?	159
Testy	161
Dlaczego testowanie kodu jest ważne?	161
Biblioteki pytest i unittest	161
Testowanie oparte na właściwościach	164
Tworzenie raportów z testów	165
Automatyzacja testów jednostkowych	166
Przygotowanie kodu do uruchomienia w środowisku produkcyjnym	166
Sprawdzanie pokrycia kodu testami	167
Program virtualenv	168
Podsumowanie	169
Dodatek	
Niezwykłe narzędzia dla języka Python	171
Sphinx	171
Coverage.py	172
pre-commit	173
Pyenv	173
Jupyter Lab	174
Pycharm/VSCode/Sublime	174
Flake8 i Pylint	175