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 umożliwia 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!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 10.

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 znajdą 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 powrócenie 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 wyrażeń. Wyrażenia przetwarzają obiekty i są osadzone w instrukcjach. Instrukcje kodują większą logikę operacji programu — wykorzystują i kierują wyrażenia 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 Pythona[1]. 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 którejś 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	`a, b, c = 'dobry', 'zły', 'paskudny'`
Wywołania	Wykonywanie funkcji	`log.write("mielonka, szynka\n")`
`print`	Wyświetlanie obiektów	`print 'The Killer', joke`
`if/elif/else`	Wybór działania	`if "python" in text:` `  print text`
`for/else`	Iteracja po sekwencjach	`for x in mylist:` `  print x`
`while/else`	Ogólne pętle	`while X > Y:` `  print 'witaj'`
`pass`	Pusty pojemnik	`while True:` `  pass`
`break, continue`	Przeskoki w pętli	`while True:` `  if not line: break`

`try/except/finally`	Przechwytywanie wyjątków	`try:`
		`  action( )`
		`except:`
		`  print 'Błąd w akcji'`

`raise`	Wywoływanie wyjątków	`raise endSearch, location`
`import, from`	Dostęp do modułów	`import sys` `from sys import stdin`

`def, return, yield`	Budowanie funkcji	`def f(a, b, c=1, *d):`
		`  return a+b+c+d[0]`
		`def gen(n):`
		`  for i in n, yield i*2`

`class`	Budowanie obiektów	`class subclass(Superclass):` `  staticData = []`

`global`	Przestrzenie nazw	`def function( ):`
		`  global x, y`
		`  x = 'nowy'`

`del`	Usuwanie referencji	`del dane[k]`
		`del dane[i:j]`
		`del obiekt.atrybut`
		`del zmienna`

`exec`	Wykonywanie łańcuchów znaków kodu	`exec "import " + modName` `exec code in gdict, ldict`
`assert`	Sprawdzanie w debugowaniu	`assert X > Y`
`with/as`	Menedżery kontekstu (Python 2.6)	`with open('data') as myfile:` `  process(myfile)`

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 pomijaniu 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 opuszczaniu 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 pożywają 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 wcinała 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 zaboleć 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 przynależeć 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 przypisania, 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 nielubiana, 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 pochwalane.

Ogólnie rzecz biorąc, nawet jeśli nie zawsze jest to wymagane, jeśli będziemy umieszczać każdą instrukcję w osobnym wierszu i zawsze wcinać 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, wciętym 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: „Wczytuj 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 załóżmy, 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 powiązanym 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!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). Przyjrzeliśmy 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.

Łamigłówka

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 zapominanie o dodaniu dwukropka na końcu wiersza nagłówka instrukcji złożonej.