Rafał Korzeniewski

Wstęp do języka Python. Część 3

Pisząc programy w Pythonie, wielokrotnie zetkniemy się z różnego rodzaju liczbami. W tym artykule przyjrzymy się dokładniej typom liczbowym, które są w nim dostępne. Zastanowimy się też, czym jest sam typ i po co się go wprowadza.

PRZYPOMNIJMY SOBIE TYPY

W poprzednich częściach dowiedzieliśmy się o istnieniu pewnych typów wbudowanych w Pythona. Zastanówmy się jeszcze chwilę, po co nam te typy w programowaniu? I co to znaczy, że pewne rzeczy są jednego typu, a drugie innego?

Idąc za słownikiem PWN, typ to „wzór, któremu odpowiada pewna seria przedmiotów, ludzi, zjawisk itp". To znaczy, że te przedmioty, ludzie czy zjawiska są do siebie podobne, są zbudowane według tego samego wzoru, mają te same cechy. Inne słowa o podobnym znaczeniu to: odmiana, rodzaj, klasa, wzór, gatunek. Słowa te pomagają nam w porządkowaniu naszej wiedzy o świecie. Możemy przy ich pomocy grupować rzeczy podobne do siebie – czyli właśnie określać ich typ. Weźmy np. środki transportu. To taki bardzo ogólny typ. Mieszczą się w nim i samoloty, i samochody, i statki, i pewnie jeszcze wiele innych typów. Wiemy, że samochody świetnie sprawdzają się na drogach. Gdy chcemy lecieć – lepiej będzie jednak skorzystać z samolotu, helikoptera czy rakiety. Różne typy nadają się do różnych rzeczy. Samochody mogą jechać do przodu, do tyłu, mogą skręcać w lewo lub w prawo. Opierają się kołami o nawierzchnię i poruszają się po niej. Samoloty natomiast, gdy lecą, mogą skręcać w prawo i w lewo, mogą się wznosić i opadać, za to raczej na pewno nie polecą do tyłu (choć prawdę mówiąc, niektóre samoloty to jednak potrafią). Zauważmy, że mówimy tu bardzo ogólnie o środkach transportu. A że to jakieś samoloty, a to jakieś samochody. O takich rzeczach mówimy, że to są abstrakcje, czyli takie uogólnione pojęcia. Konkretne pojazdy, np. twój rower, jeep mamy, czy skuter taty, to są pojedyncze egzemplarze danej klasy. W programowaniu mówimy często, że są to instancje danego typu czy danej klasy. Podsumowując, typ to jest pewna abstrakcja, uogólniony przepis, a instancja to będzie konkretny przykład zastosowania tego typu.

W części drugiej wprowadzenia do Pythona dowiedzieliśmy się, że do obiektów możemy przypisywać nazwy. Jeśli popatrzymy na same nazwy, to nie zawsze zrozumiemy, jakiego typu wartości się za nimi kryją. Do zmiennej `x` możemy w końcu przypisać zarówno napis, jak i jakąś liczbę. A to dwa zupełnie inne typy. Jak więc rozpoznawać te typy? W Pythonie służyć do tego może funkcja `type`. Wywołana z jednym argumentem zwróci nazwę jego typu. W niektórych sytuacjach w tej nazwie można zobaczyć słowo `class` (np. jeśli użyjemy jej w REPL a nie w IPython). Oznacza ono, że coś jest jakiejś klasy. W dawnych wersjach Pythona istniało silne rozróżnienie między `type` i `class`. Wraz z kolejnymi wersjami te różnice się zacierały. Dziś – poza bardzo zaawansowanymi zagadnieniami, takimi jak metaklasy, możemy uznać, że te słowa będą oznaczać to samo. A więc możemy powiedzieć, że coś ma określony typ albo jest określonej klasy. Obok funkcji `type` istnieje jeszcze jedna funkcja – `isinstance`, która pozwala na testowanie, czy coś jest instancją danej klasy. Przyjmuje ona dwa argumenty i sprawdza, czy ten pierwszy argument jest instancją klasy podanej jako drugi argument. W ramach ćwiczenia spróbuj samodzielnie sprawdzić dokumentację tych funkcji (użyj funkcji `help`). Omawiane funkcje możemy użyć tak jak w Listingu 1.

Listing 1. Przykład użycia funkcji isinstance

```
In [1]: type(10)
Out[1]: int

In [2]: type("10")
Out[2]: str

In [3]: isinstance(10, int)
Out[3]: True

In [4]: isinstance(10, str)
Out[4]: False
```

Zwróć uwagę na nazwę funkcji `isinstance`. Po polsku można spróbować odczytać ją dosłownie tak: czy `10` jest instancją klasy `int`? Na tak postawione pytanie poprawną odpowiedzią jest: tak. Czy `10` jest instancją `str`? Python mówi nam, że nie. Oczywiście zamiast tej `10` mogłaby w tym miejscu być wstawiona nazwa zmiennej, do której `10` jest przypisane. Nie miałoby to wpływu na działanie. Zapamiętaj te funkcje – jeszcze nam się przydadzą. Teraz przyjrzymy się nieco dokładniej wbudowanym w Pythona typom liczbowym.

TYP INT

Dlaczego warto mieć osobny typ na liczby? Chociażby po to, by wygodnie na nich pracować i wykonywać obliczenia. Jeśli zobaczymy między dwiema liczbami znak +, to będziemy wiedzieć, że trzeba je do siebie dodać, a nie np. skleić. W szkole mamy osobny przedmiot, na którym uczymy się rachować, czyli liczyć. Tam też korzystamy z tych unikalnych cech liczb, które pozwalają nam przy pomocy różnych symboli przeprowadzać obliczenia. Nie inaczej będzie w programowaniu. Do czego nadają się liczby całkowite? Na przykład do tego, by coś ponumerować. Czy możemy sobie numerować dowolnie długo? Czy może kiedyś numerki nam się skończą? W świecie matematyki – dowolnie długo. Czy tak samo jest w programowaniu? Nie. Komputery to urządzenia, które operują na bitach. Bit to taka podstawowa porcja informacji i ma tylko dwie wartości: 0 lub 1, czyli stan włączony (stan wysoki) lub wyłączony (stan niski). Komputery to fizyczne urządzenia i muszą w jakiś fizyczny sposób takie informacje zapisywać, przechowywać. Wykorzystywano do tego różne sposoby: wypustki na bębnach, dziurki w kartach, ładunki elektryczne, czy pole magnetyczne. A więc dziurka jest lub jej nie ma. Ładunek jest lub go nie ma. Obszar jest namagnesowany lub nie. Oczywiście ten opis jest znacznie uproszczony. W rzeczywistości bowiem i pole magnetyczne, i ładunki elektryczne czy inne parametry fizyczne nie są zerowe. By lepiej oddawać tę rzeczywistość mówimy wtedy o stanach niskich i wysokich. Istotą jest jednak to, by można było w jakiś sposób wyróżnić dwa stany. Takie dwie wartości mogą służyć do zapisu liczb w systemie dwójkowym. W systemie tym są tylko dwie cyfry: 0 i 1. Cyfry te ustawione są obok siebie w pewnym porządku. Każda kolejna pozycja to kolejna potęga dwójki (porównaj artykuł: „Dwójkowe, dziesiętne, szesnastkowe" Witolda Wrotka z numeru 1/2019). Potęgi, jak zapewne wiesz, oznaczają ile razy jakaś liczba ma być przemnożona przez samą siebie. Na przykład 2^2 to jest to samo co 2 razy 2.

$2^2 = 2 \cdot 2 = 4$
$2^3 = 2 \cdot 2 \cdot 2 = 8$
$2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16$

i tak dalej.

CIEKAWOSTKA

Ilustracja 1. Carl Friedrich Gauss (źródło: https://pl.wikipedia.org/wiki/Carl_Friedrich_Gauss#/media/Plik:Carl_Friedrich_Gauss.jpg)

Python ma jeszcze jeden typ liczbowy – `complex`, czyli liczby zespolone. Są to liczby składające się z dwóch części, które można zapisać jako `x + yj`, gdzie j to specjalna liczba, która podniesiona do kwadratu daje -1. Natomiast x i y to liczby rzeczywiste, przy czym x nazywana jest częścią rzeczywistą, a y – częścią urojoną. Do matematyki w takiej postaci wprowadził je w 1832 roku niemiecki matematyk Carl Friedrich Gauss. Sama koncepcja pojawiła się jednak znacznie wcześniej. Pierwiastek kwadratowy z liczby ujemnej mógł być rozważany już w starożytności przez Herona z Aleksandrii. Nazwa „liczby urojone" pojawiła się z kolei w pracy Kartezjusza z 1637 roku. Z kolei inny matematyk – Girolamo Cardano, nazywał w XVI wieku takie liczby liczbami fikcyjnymi.

Natomiast jeśli jakąś dodatnią liczbę podniesiemy do potęgi 0, to wynikiem będzie zawsze 1, na przykład:

$2^0 = 1$ i np. $123^0 = 1$

W systemie dziesiętnym, tak naturalnym dziś dla większości ludzi, każda liczba może być zapisana przy pomocy pewnej kombinacji dziesięciu cyfr: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Cyfry te ułożone są w pewnym porządku. Ta ostatnia – licząc od lewej strony – oznacza jedności, ta poprzedzająca ją to będą dziesiątki, potem setki, tysiące itd. Możemy to zapisać następująco:

$$123 = 100 + 20 + 3 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

Jak więc widzimy, kolejne cyfry oznaczają, ile razy występuje dana potęga dziesiątki.

W systemie dwójkowym zasada jest taka sama. Tyle że mamy dwie cyfry: 0 i 1 i oznaczają one kolejne potęgi dwójki.

Jeśli zachowamy tę samą kolejność, to ta najbardziej po prawej oznacza 2^0. Jeśli do zapisu liczby użyjemy 8 bitów, to możemy przy ich pomocy zapisać liczby dziesiętne z zakresu od 0 do 255. Na przykład liczba:

$$01001100_{\text{Binarnie}} =$$
$$0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 =$$
$$0 + 64 + 0 + 0 + 8 + 4 + 0 + 0 = 76_{\text{Dziesiętnie}}$$

W obu przypadkach to nadal ta sama liczba, tyle że przedstawiona w inny sposób, w innym systemie liczbowym. Robiąc tak dla każdej liczby, otrzymamy:

$$0_{\text{Dziesiętnie}} = 0000\ 0000_{\text{Binarnie}}$$
$$1_{\text{Dziesiętnie}} = 0000\ 0001_{\text{Binarnie}}$$
$$2_{\text{Dziesiętnie}} = 0000\ 0010_{\text{Binarnie}}$$
$$3_{\text{Dziesiętnie}} = 0000\ 0011_{\text{Binarnie}}$$
$$\ldots$$
$$255_{\text{Dziesiętnie}} = 1111\ 1111_{\text{Binarnie}}$$

8 bitów to jeden bajt. Tyle różnych liczb możemy zapisać przy pomocy jednego bajta. By zapisać większą liczbę, musielibyśmy użyć większej ilości bitów. Niektóre języki wprowadzają dla liczb całkowitych typy, które mają określoną liczbę bitów. Na przykład: `int8` będzie miał 8 bitów – tak jak w powyższym przykładzie, `int32` będzie miał już 32 bity, a więc pozwoli na zapisanie znacznie większych liczb. W Pythonie – standardowo – mamy jeden typ dla liczb całkowitych, nazywa się on po prostu `int` i można w nim zapisać teoretycznie dowolnie wielką liczbę. Ograniczają nas jedynie możliwości sprzętowe.

Skoro te same liczby można reprezentować na różne sposoby, przy użyciu różnych systemów liczbowych, to czy w Pythonie możemy z takich systemów korzystać? Czy on je zrozumie? Okazuje się, że w wielu przypadkach tak jest. Standardowo Python wypisuje i przyjmuje liczby w systemie dziesiętnym (*decimal*). Ale poradzi też sobie z systemem dwójkowym (*binary*), ósemkowym (*octal*) i szesnastkowym (*hexadecimal*). Musimy mu tylko powiedzieć, o jaki system chodzi. Jak to zrobić? Służą do tego przedrostki. W Tabeli 1 znajdziesz przykłady takich reprezentacji. Ważne jest, by stosować odpowiednie dla danego systemu cyfry.

SYSTEM	PREFIX	PRZYKŁADY	DOSTĘPNE CYFRY
dziesiętny		`10, 123`	`0,1,2,3,4,5,6,7,8,9`
binarny	`'0b' lub '0B'`	`0b111, 0B10101`	`0,1`
ósemkowy	`'0o' lub '0O'`	`0o77, 0O12`	`0,1,2,3,4,5,6,7`
szesnastkowy	`'0x' lub '0X'`	`0xff, 0X1a`	`0,1,2,3,4,5,6,7,8,9, a,b,c,d,e,f`

Tabela 1. Reprezentacja liczb w wybranych systemach liczbowych

Przykłady znajdziesz też w Listingu 2. Użycie prefixu pozwala na podanie liczby w odpowiednim systemie. Wartość prezentowana jest w systemie dziesiętnym. Zachęcam cię do eksperymentów we własnym zakresie.

Listing 2. Przykłady liczb w różnych systemach

```
In [1]: 0b111
Out[1]: 7
In [2]: 0o77
Out[2]: 63
In [3]: 0xff
Out[3]: 255
```

Jeśli chcemy wykonać operację odwrotną, tzn. poznać, jak liczba dziesiętna reprezentowana będzie w innym systemie, to możemy skorzystać z jednej z wbudowanych

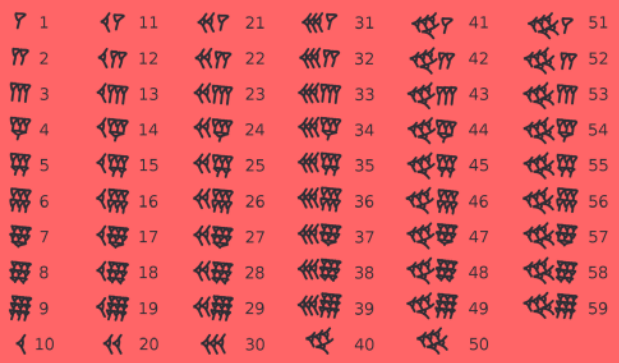
w Pythona funkcji: `bin`, `oct`, `hex`. Tak jak na Listingu 3. Zwracana wartość jest wtedy napisem.

Listing 3. Przykłady przedstawiania liczb dziesiętnych w różnych systemach liczbowych

```
In [4]: bin(12)
Out[4]: '0b1100'
In [5]: oct(342391)
Out[5]: '0o1234567'
In [6]: hex(12648430)
Out[6]: '0xc0ffee'
```

WARTO WIEDZIEĆ

O ile wygodnie jest konstruować komputery w oparciu o system dwójkowy, o tyle ludzie w swoim codziennym życiu posługują się systemem dziesiętnym. W przeszłości jednak także inne systemy liczbowe były bardzo popularne. Jednym z nich był system sześćdziesiątkowy stosowany w starożytnym Babilonie. Jego pozostałości mamy do dziś np. w określaniu czasu.



Ilustracja 2. Cyfry babilońskie (źródło: https://en.wikipedia.org/wiki/Babylonian_cuneiform_numerals#/media/File:Babylonian_numerals.svg)

W systemie tym mamy 59 cyfr, które w zasadzie zbudowane są z dwóch symboli – pionowego klina 𒁹 dla jedności i poziomego 𒌋 dla dziesiątek. Do zapisu większych liczb używano tych samych symboli. Stąd jeden pionowy klin mógł oznaczać zarówno 1, jak i 60 czy 3600. Jest to system pozycyjny, ale brakuje w tym systemie cyfry 0 – prawdziwą wartość odczytywano więc na podstawie kontekstu lub obliczeń, choć znane są też tabliczki, na których pozostawiano puste miejsce jako oznaczenie braku na tej pozycji. System oparty na liczbie 60 ma bardzo wiele dzielników, są to liczby 2, 3, 4, 5, 6, 10, 12, 15, 20 i 30. W związku z tym wiele operacji wymagających dzielenia dawało całkowite wyniki i łatwiej było to policzyć w pamięci.

Wspomniałem wcześniej, że `int` to nie tylko typ, ale i wbudowana funkcja. A my wiemy już, że funkcje można wywoływać. Co się stanie, jeśli wywołam tę funkcję? Poeksperymentujmy:

Listing 4. Przykłady wywołania funkcji int. Operacje zmieniające typ nazywamy rzutowaniem typu

```
In [1]: int()
Out[1]: 0
In [2]: int(10)
Out[2]: 10
In [3]: int(2.7)
Out[3]: 2
In [4]: int(True)
Out[4]: 1
In [5]: int("101")
Out[5]: 101
In [6]: int("a")
---------------------------------------------------------
ValueError                Traceback (most recent call last)
<ipython-input-6-d9136db7b558> in <module>
----> 1 int("a")

ValueError: invalid literal for int() with base 10: 'a'
```

Widzimy, że w większości powyższych przypadków ta funkcja zwróciła sensowne wyniki. Wywołana bez argumentu zwróciła po prostu 0. Gdy podałem jej liczbę całkowitą, zwróciła tę liczbę. Liczbę zmiennoprzecinkową zamieniła w jej całkowitą część. Wartość `True` zamieniła na `1`. A napis reprezentujący liczbę na tę liczbę. Nie udało się jednak zamienić litery „a” na liczbę, co w sumie nie powinno dziwić. Z dokumentacji dowiemy się też, że możemy tę funkcję wykorzystać przy konwersji na inny system liczbowy. Niekoniecznie musi to być jeden z systemów, które już poznaliśmy. Powiedzmy, że mamy liczbę w systemie trójkowym zapisaną jako 222. Chcąc się dowiedzieć ile, to w systemie dziesiętnym musielibyśmy wykonać rachunek:

$$2 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 2 \cdot 9 + 2 \cdot 3 + 2 \cdot 1 = 18 + 6 + 2 = 26$$

Podobnie moglibyśmy się zastanawiać, jaką liczbę w systemie dziesiętnym reprezentuje liczba zapisana w następujący sposób: `'c0ffee'` w systemie szesnastkowym.

Potrafimy już to policzyć, ale prościej będzie skorzystać z funkcji `int`:

Listing 5. Przykład konwersji napisów reprezentujących liczby z systemu trójkowego i szesnastkowego na system dziesiętny. Argument base w pierwszym przypadku podany jest wprost. W drugim przypadku Python domyśli się o jaki argument chodzi na podstawie kolejności podanych argumentów

```
In [12]: int("222", base=3)
Out[12]: 26
In [13]: int('c0ffee', 16)
Out[13]: 12648430
```

Dla porządku warto dodać jeszcze, że w Pythonie możemy skorzystać z dodatkowej, ułatwiającej czytanie notacji, w której cyfry oddzielamy od siebie znakiem _. Możemy napisać 1_000_000_123 zamiast 1000000123, 0b0001_1001 zamiast 0b00011001 itd, itd.

Skoro wiemy już, co może być przechowywane w danym typie, zastanówmy się jeszcze, jakie operacje możemy na tych wartościach wykonać. Po pierwsze, operacje arytmetyczne, takie jak: dodawanie, odejmowanie, mnożenie, dzielenie czy potęgowanie. Do tego dochodzi jeszcze dzielenie całkowite oraz reszta z dzielenia, czyli tzw. dzielenie modulo. Tym operacjom odpowiadają w Pythonie odpowiednie operatory, czyli symbole, znaki. Zobaczmy to na przykładach:

Listing 6. Podstawowe operacje i operatory arytmetyczne na liczbach

```
In [1]: 1 + 1   # dodawanie
Out[1]: 2
In [2]: 3 - 1   # odejmowanie
Out[2]: 2
In [3]: 4 * 2   # mnożenie
Out[3]: 8
In [4]: 10 / 2  # dzielenie
Out[4]: 5.0
In [5]: 10 ** 2 # potęgowanie
Out[5]: 100
In [6]: 10 // 3 # dzielenie całkowite
Out[6]: 3
In [7]: 10 % 3  # dzielenie modulo
Out[7]: 1
```

Dzielenie całkowite zwraca część całkowitą z dzielenia. Ucina natomiast część dziesiętną. Korzystając z rozwinięcia dziesiętnego, jeśli 3 podzielimy przez dwa, to dostaniemy 1.5 (stosuję tu kropkę, jak w Pythonie, a nie przecinek jak na zajęciach matematyki). Część całkowita z tej liczby to 1. Dzielenie całkowite zwraca właśnie tę część. Na przykład `3 // 2` da nam 1, natomiast `7 // 3` da nam 2, podobnie jak `9 // 4`.

Dzielenie modulo (%) z kolei zwróci nam resztę z dzielenia, np. 6 % 3 daje nam 0, bo 6 dzieli się bez reszty przez 3. 5 % 3 da nam 2, ponieważ w 5 mieści się jedna 3. Jeśli od 5 odejmiemy tę 3, to zostaje nam liczba 2, która jest już mniejsza od 3. 10 % 3 też da nam 1. W 10 mieszczą się trzy 3, które razem dają 9. Jak od 10 odejmiemy 9, to zostanie 1.

Oprócz operacji arytmetycznych na liczbach często też chcemy je ze sobą porównywać. Do takich operacji służą operatory porównania pokazane w Listingu 7.

Listing 7. Operatory porównania

```
In [12]: x = 10
In [13]: y = 20
In [14]: x == y    # czy równe?
Out[14]: False
In [15]: x < y    # czy x jest mniejsze niż y?
Out[15]: True
In [16]: x > y    # czy x jest większe niż y?
Out[16]: False
In [17]: x <= 10  # czy x jest mniejsze lub równe 10
Out[17]: True
In [18]: y >= 20  # czy y jest większe lub równe 20
Out[18]: True
In [19]: x != y   # czy x jest różne od y
Out[19]: True
```

Jak widzimy, wynikiem działania takich operatorów jest zawsze jakaś wartość logiczna. Wspominałem o nich w pierwszej części wprowadzenia do Pythona. Wrócimy jeszcze do tych wartości w tym artykule.

Czasem, szczególnie gdy używamy nazw zmiennych, może okazać się, że próbujemy porównać ze sobą jakieś obiekty, których komputer porównać nie potrafi. Zobaczmy to w Listingu 8:

Listing 8. Próba porównania dwóch wartości należących do różnych typów

```
In [1]: 10 > "a"
---------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-1-5582870fa9b0> in <module>
----> 1 10 > "a"

TypeError: '>' not supported between instances of
'int' and 'str'
```

Próbowałem sprawdzić, czy 10 jest większe niż litera „a". Python odpowiedział mi komunikatem błędu, który mówi, że operacja > (większy niż) dla tych dwóch różnych typów nie jest wspierana. W przyszłości będziemy tworzyć własne typy i będziemy uczyć Pythona, jak je ze sobą lub innymi typami porównywać.

W przypadku liczb możemy też wykorzystywać uproszczone operatory przypisania. Na przykład a += 1 można zapisać też jako a = a + 1. Sprawdź to w ramach ćwiczeń także dla innych, podobnych operatorów: -=, *=, /=, //=, %=, **=.

Typ bool

W poprzedniej części artykułu wspomnieliśmy, że w Pythonie mamy specjalny typ dla wartości logicznych. Nazwa tego typu – `bool` wzięła się od George'a Boole'a, twórcy algebry Boole'a – dziedziny matematyki, która dała podstawy do wprowadzenia logiki w komputerach. W tym miejscu spojrzymy jednak na ten typ jak na liczby. A logiką zajmiemy się w jednym z następnych artykułów. Typ `bool` podobnie jak `int` jest podtypem specjalnego typu liczbowego (`numbers.Integral`). Co to dokładniej znaczy, dowiemy się w późniejszym czasie. Teraz przyjmijmy że są to rzeczy do siebie podobne. Podtyp przejmuje jakieś cechy typu. O ile w `int` możemy zapisać dowolne liczbowe wartości, o tyle w `bool` są tylko dwie: `True` i `False`. Zachowują się one odpowiednio jak 1 i 0. Możemy to sprawdzić, wykonując proste operacje arytmetyczne.

Listing 9. Wartości logiczne mają liczbową reprezentację

```
In [1]: 1 + True
Out[1]: 2
In [2]: False - 2
Out[2]: -2
In [3]: 3 / True
Out[3]: 3.0
In [4]: 3 / False
---------------------------------------------------------
ZeroDivisionError   Traceback (most recent call last)
<ipython-input-4-f0afc627b943> in <module>
----> 1 3 / False

ZeroDivisionError: division by zero
```

Jak widzimy, `True` i `False` mogą zachowywać się jak liczby 1 i 0. Zobaczmy jednak, że są też jednocześnie czymś innym:

Listing 10. 1 i True oraz 0 i False to różne obiekty

```
In [1]: 1 == True
Out[1]: True
In [2]: 1 is True
Out[2]: False
In [3]: 0 == False
Out[3]: True
In [4]: 0 is False
Out[4]: False
```

CIEKAWOSTKA

Na terenie Polski, jeszcze za naszych dziadków czy pradziadków, posługiwano się specjalnymi liczbami na oznaczenie ilości. Opierały się one na liczbach 12 i 60. Tuzin to właśnie 12. Kopa to 60, czyli 5 tuzinów. Mendel to 15, czyli jedna czwarta kopy. Gros natomiast to 144, czy 12 x 12, a więc tuzin tuzinów. Ludziom od dawna przydają się liczby i mamy wiele sposobów na to, by je opisywać i dobrze sobie z nimi radzić.

Typ Float

Ostatnim typem liczbowym, który omówimy w tym artykule, jest typ `float`. Dlaczego potrzebny jest osobny typ na liczby zmiennoprzecinkowe? Przecież liczby to liczby? Okazuje się jednak, że te liczby, które nie są całkowite, stanowią dla komputera większe wyzwanie.

Obejmują one bowiem trochę trudniejsze przypadki, a mianowicie takie, których nie da się zapisać przy pomocy skończonej liczby cyfr. I nie muszą to być jakieś nieskończenie wielkie liczby. Przykładem może być liczba π. Ze szkoły wiesz pewnie, że wynosi ona około 3,14. No właśnie to „około" jest tutaj kluczowe. Tak naprawdę, po przecinku jest w tej liczbie nieskończenie wiele cyfr. Poznajemy je coraz lepiej, ale i tak nie znamy wszystkich. Przedstawiając π przy pomocy cyfr, stosujemy zawsze jakieś przybliżenie. Wyobraź sobie liczbę 1/3. Moglibyśmy ją zapisać jako 0,(3) – czyta się to jako 0 i 3 w okresie, co oznacza, że ta 3 po przecinku powtarza się nieskończenie wiele razy, czyli liczba ta ma rozwinięcie okresowe, które w zasadzie nigdy się nie kończy. Łatwo to zapisać na kartce, ale trudniej jest nauczyć tego komputera. Jak sobie poradzi z tym Python?

Listing 11. Liczba zmiennoprzecinkowa ma swoją dokładność

```
In [1]: 1/3
Out[1]: 0.3333333333333333

In [2]: 10/3
Out[2]: 3.3333333333333335
```

W Listingu 11, szczególnie w sekcji `out[2]`, widzimy, że do pewnego miejsca to rozwinięcie dziesiętne jest dokładne, a później coś się niestety psuje. Wiemy już, że współczesne komputery posługują się bitami do zapisu wartości. A bit to 1 lub 0. Nie inaczej będzie z liczbami typu `float`. Na przykład liczbę 0.125 możemy rozłożyć w systemie dziesiętnym na taką sumę:

$$\frac{1}{10} + \frac{2}{100} + \frac{5}{1000} = \frac{1}{8} = \frac{1}{10^1} + \frac{2}{10^2} + \frac{1}{10^3}$$

W systemie binarnym zapisać możemy tę liczbę jako:

$$0.001 = \frac{0}{2} + \frac{0}{4} + \frac{1}{8} = \frac{0}{2^1} + \frac{0}{2^2} + \frac{1}{2^3}$$

W tym przypadku szczęśliwie reprezentacja może być dokładna, zapisana przy pomocy kilku cyfr. Ale jak zapisać w systemie dwójkowym liczbę dziesiętną 0.1?

$0.1_{Decimal}$ = 0.0001100110011001100110011001100110011001100110011 00110011...$_{Binary}$ = $0.0(0011)_{Binary}$

Mamy tu więc stosunkowo prostą liczbę w systemie dziesiętnym, która ma nieskończone, okresowe rozwinięcie w systemie binarnym. Nie da się więc w tym systemie, przy pomocy skończonej liczby cyfr, przedstawić takiej liczby. Takich problemów napotkamy bardzo wiele. Dlatego ogranicza się liczbę cyfr, których możemy użyć do ich zapisu, tak by nasze komputery miały szansę wykonywać operacje na takich liczbach i się przy tej okazji nie zawieszać. Jest to więc zapis przybliżony. A przybliżenia można osiągać na wiele sposobów. Stosując różne algorytmy, możemy uzyskiwać różne wyniki. By uniknąć chaosu, wprowadza się więc standardy. Obecnie większość urządzeń pracuje w oparciu o standard IEEE 754. Obejmuje on liczby pojedynczej i podwójnej precyzji. Liczby typu `float` zapisane w Pythonie rozumiane są przez zdecydowaną większość platform sprzętowych jako liczby w standardzie IEEE 754 podwójnej precyzji. Opis tego standardu wychodzi daleko poza ramy tego artykułu. Niemniej jednak warto wiedzieć, że coś takiego istnieje. Zachęcam do samodzielnego zapoznania się ze sposobem operowania na liczbach w tym systemie. W Listingu 12 znajdziesz za to kilka ciekawostek – zaskakujących wyników, które z tego standardu wynikają. Nie są to bynajmniej błędy, ale konsekwencja zastosowanych przybliżeń. Zastosowano tu też notację naukową.

Listing 12. Zaskakujące wyniki reprezentacji i działań na liczbach zmiennoprzecinkowych

```
In [1]: 1.79e308
Out[1]: 1.79e+308

In [2]: 1.8e308
Out[2]: inf

In [3]: 5e-324
Out[3]: 5e-324

In [4]: 1e-325
Out[4]: 0.0

In [5]: 1.0 == 1.00000000000000001
Out[5]: True

In [6]: 0.1 + 0.1 + 0.1 == 0.3
Out[6]: False
```

Notacja naukowa, o której wspomniano, przydaje się szczególnie wtedy, gdy mamy do czynienia z liczbami

bardzo dużymi lub bardzo małymi. Na przykład zamiast pisać 0.00000001 możemy napisać $1e^{-8}$. Czasem Python robi to za nas:

Listing 13. Notacja naukowa

```
In [2]: 0.00000001
Out[2]: 1e-08
In [3]: 10e-12
Out[3]: 1e-11
In [4]: 101e-12
Out[4]: 1.01e-10
In [5]: 4e15
Out[5]: 4000000000000000.0
In [6]: 4e16
Out[6]: 4e+16
```

Wiemy już, że na liczby typu `float` musimy w Pythonie szczególnie uważać. Czasem nawet proste zadania, jak te z końcówki Listingu 12, mogą dawać nieprawidłowe wyniki. Są oczywiście sposoby, by sobie z takimi problemami radzić. Można skorzystać z zaokrąglania liczb przy pomocy funkcji `round`, choć tu też nie każde użycie da oczekiwany wynik, należy bowiem zaokrąglić wynik sumowania, a nie każdą z pojedynczych liczb:

```
In [1]: round(.1, 1) + round(.1, 1) + round(.1, 1) ==
round(.3, 1)
Out[1]: False

In [2]: round(.1 + .1 + .1, 10) == round(.3, 10)
Out[2]: True

In [3]: round(.1 + .1 + .1, 1) == round(.3, 1)
Out[3]: True
```

Można też skorzystać z modułu `decimal`, który dostarcza specjalny typ `Decimal`, przy pomocy którego możemy robić precyzyjne obliczenia – pod warunkiem poprawnego użycia `Decimal`:

Listing 14. Użycie modułu decimal do precyzyjnych obliczeń na liczbach zmiennoprzecinkowych

```
In [1]: from decimal import Decimal
In [2]: Decimal(.1)
Out[2]:
Decimal('0.1000000000000000055511151231257827021181
583404541015625')
In [3]: Decimal(.3)
Out[3]:
Decimal('0.2999999999999999888977697537484345957636
83319091796875')
In [4]: Decimal(.1) + Decimal(.1) + Decimal(.1) ==
Decimal(.3)
Out[4]: False
In [5]: Decimal(".1")
Out[5]: Decimal('0.1')
In [6]: Decimal(".3")
Out[6]: Decimal('0.3')
In [7]: Decimal("0.1") + Decimal("0.1") +
Decimal("0.1") == Decimal("0.3")
Out[7]: True
```

Na liczbach zmiennoprzecinkowych można oczywiście wykonywać wszystkie te operacje, które są dostępne dla liczb całkowitych. Stosujemy te same operatory i zasady. Analogicznie też jak w poprzednich przypadkach mamy dostępną funkcję `float`, której działanie z pewnością zechcesz samodzielnie przetestować. Przykład użycia modułu `decimal` wybiega mocno w przód – nie chciałem jednak pozostawić cię w przekonaniu, że w Pythonie nie da się wykonać prawidłowo tak prostych operacji, jak te z Listingu 12.

Na dziś wystarczy. Dowiedzieliśmy się dużo nowego o Pythonie. Zapoznaliśmy się z podstawowymi typami liczbowymi dostępnymi w tym języku. Mam nadzieję, że część z tych rzeczy cię zaskoczyła, skłoniła do testowania, zadawania pytań i własnych poszukiwań.

W następnych artykułach będziemy kontynuować naszą przygodę. Mam nadzieję, że pozostaniesz z nami!

Rafał Korzeniewski

Z wykształcenia muzyk-puzonista i fizyk. Z zamiłowania i zawodu Pythonista. Trener Pythona w ALX, współorganizator PyWaw (http://pywaw.org) – warszawskiego meetupu poświęconego Pythonowi. W wolnych chwilach uczy się gry na nowych instrumentach, udziela się społecznie i dużo czyta.

KORZENIEWSKI@GMAIL.COM