https://printpython.pl/python/magiczne-metody-czyli-szczypta-magii-w-pythonie/

Moim zdaniem rzeczą, która należy wymienić na początku wśród tych, które sprawiają, że Python jest tak przyjazny poczatkującym użytkownikom jest jego wszechobecna spójność. Już po chwili spędzonej z tym językiem, jesteśmy w stanie domyślić się jak będą działały rzeczy, których jeszcze nie poznaliśmy, tylko na podstawie dotychczasowych doświadczeń. Za przykład może nam posłużyć zadanie sprawdzenia liczby elementów w dowolnej kolekcji. W taki sam sposób jak dla stringa możemy zrobić to funkcją `len()` w przypadku listy, czy też bardziej zaawansowanej kolekcji jak na przykład OrderedDict. Możemy sprawić, że w ten sam sposób zachowywać będą się klasy stworzone przez nas, dzięki wykorzystaniu magicznych metod.

Magiczne metody (inaczej atrybuty specjalne klasy) to metody, które pozwalają na dodanie do definiowanych przez nas klas Pythonowego interfejsu. Maja one ściśle określone nazwy (zawsze otoczone podwójnymi podkreślnikami) odpowiadające poszczególnym zadaniom. Zadania te, wykonywane przez magiczne metody, możemy podzielić na kilka typów i w tym artykule chciałbym przedstawić najpopularniejsze metody dla wybranych typów. Opis wszystkich metod magicznych dostępnych w Pythonie znajdziesz w dokumentacji [https://docs.python.org/3/reference/datamodel.html]

Tworzenie i usuwanie instancji

`__new__(cls, ...)`

Jest to metoda wywoływana jako pierwsza podczas tworzenia nowych obiektów. Jest wywoływana jeszcze przed najpopularniejszą z magicznych metod czyli `__init__` i jej zadaniem jest stworzenie i zwrócenie nowej instancji danej klasy, która to zaraz potem jest przekazywana do metody `__init__` jako pierwszy argument (self).

`__init__(self, ...)`

Tak jak wspomniałem powyżej to najbardziej znana z magicznych metod. Spotkał się z nią każdy kto ma jakiekolwiek doświadczenie z programowaniem obiektowym w Pythonie. Często nazywana jest konstruktorem, którym tak właściwie jest razem z metodą `__new__`. Jej zadaniem jest inicjalizacja obiektu z pomocą argumentów przekazanych przy wywołaniu klasy.

`__del__(self)`

Po drugiej stronie od powyższych metod jest metoda `__del__`. Wywoływana jest ona w momencie gdy obiekt kończy swój cykl życia i za zadanie ma zrobienie porządku po nim. Należy pamiętać, że metoda ta wywoływana jest gdy licznik referencji do obiektu spadnie do 0, co oznacza, że nie nastąpi to zawsze gdy wywołamy `del x`.

Reprezentacja obiektów

`__str__(self)`

Jest to metoda wywoływana przy konwersji naszego obiektu do stringa `str(x)` , oraz przy

wywoływaniu funkcji `print(x)`. Jeżeli ją implementujemy to powinna ona zwracać string z czytelną dla ludzi reprezentacją naszego obiektu.

`__repr__(self)`

Ta metoda natomiast powinna zwrócić oficjalną reprezentację, jej wynik możemy zobaczyć wywołując funkcję `repr()`, albo wpisując nazwę naszego obiektu w linii poleceń interpretera. Dobrą praktyką jest zwracanie poprawnego wyrażenia pozwalającego na ponowne stworzenie takiego samego obiektu.

`__hash__(self)`

Metoda `__hash__` wykorzystywana jest do haszowania naszego obiektu, czyli zamiany jego wartości na wartość liczbową. Jej implementacja jest potrzebna gdy chcemy ustalić szczególne zachowanie naszych obiektów w roli kluczy słowników. Jeżeli nie chcemy by nasz obiekt był haszowalny, tym samym nie mógł być kluczem słownika, należy ustawić w klasie wartość `__hash__ = None`. To samo dzieje się automatycznie, również gdy w naszej klasie deklarujemy wyłącznie metodę `__eq__`. Jeżeli chcemy dostosować zachowanie obiektu jako klucz słownika (na przykład żeby brane pod uwagę było konkretne pole z obiektu) powinniśmy zaimplementować obie te metody.

Konwersja typu

Tę kategorię pozwolę sobie potraktować zbiorczo (bo chyba każdy domyśla się do czego służą metody `__int__` i `__float__`) po to żeby szybko przejść do tych najciekawszych metod.

A jakby ktoś jednak nie wiedział to metody te służą do konwersji na jeden z wbudowanych typów Pythona. I jedyne o czym należy pamiętać to to, żeby zwracały one wartość odpowiedniego typu.

W tej kategorii ciekawostkę stanowi metoda `__bool__` jej implementacja powinna zwracać wartości `True` albo `False`. Jeżeli jej nie zaimplementujemy to wartość wyrażenia `bool(x)` zostanie obliczona na podstawie wyniku metody `__len__`, jeżeli obie metody nie zostaną zaimplementowane to nasz obiekt zawsze będzie miał wartość `True`.

Metody typowe dla kontenerów

Chcąc stworzyć klasę, która będzie kontenerem w pojęciu Pythonowym, będziemy potrzebowali zaimplementować w niej kilka metod typowych dla kontenerów. Podstawowymi będą `__len__` i `__getitem__`, dodatkowo gdy chcemy by nasze obiekty były mutowalne będziemy potrzebowali metod `__setitem__` i `__delitem__`.

`__len__(self)`

Metoda potrzebna do obliczenia wartości przy wywołaniu `len(x)` dla naszego obiektu. Powinna zwracać liczbę całkowitą większa lub równą 0.

`__getitem__(self, key)`

Implementacja tej metody pozwala na wywołania typu `x[klucz]`. Dla sekwencji, czyli typów danych podobnych do listy czy tupli, klucz może mieć wartość liczby całkowitej (również ujemnej), oraz być obiektem typu slice – dla wywołań takich jak na przykład `x[1:5]`, które zamienione zostanie na `x[slice(1, 5, None)]`. Warto pamiętać też o wyjątkach zwracanych gdy podany klucz jest niepoprawny. `TypeError` gdy klucz jest niewłaściwego typu (np. nie jest liczbą całkowita dla

sekwencji), `KeyError` gdy klucz nie znajduje się w kontenerze (zwłaszcza dla typów mapujących, takich jak słownik), oraz `IndexError` gdy indeks jest spoza dostępnego zakresu. Ten ostatni wyjątek jest o tyle istotny, że jest wartością oczekiwaną przez pętlę `for` podczas iterowania po sekwencji.

`__setitem__(self, key, value)`

Metoda ta pozwala na operację przypisania do obiektu w kontenerze. Powinna być zaimplementowana jedynie wtedy gdy chcemy, by obiekty w naszym kontenerze miały możliwość podmieniania lub dodawania nowych dla typów mapujących. Pozostałe zalecania są identyczne jak w przypadku `__getitem__`.

`__delitem__(self, key)`

Metoda ta pozwala na operację usuwania obiektów w kontenerze. Zaimplementuj ją jedynie wtedy gdy chcesz, by obiekty wewnątrz były usuwalne poprzez `del x[klucz]`. Pozostałe zalecania są identyczne jak w przypadku `__getitem__`.

Powyższe metody powinny zapewnić nam podstawowy interfejs dla obiektu typu kontener. Warto mieć w pamięci jednak również to, że porządny kontener zapewnia też interfejs w postaci kilku niemagicznych metod takich jak na przykład: `append()`, `index()`, `insert()`, `keys()`, `pop()`, `values()`. A kontenery z wyższej półki dają możliwość używania operatorów dodawania i mnożenia, ale o tym powiemy sobie za chwilę.

Tworząc swój własny kontener danych warto też zajżeć do modułu `collections.abc` i wybrać jedną z dostępnych tam klas jako naszą klasę bazową.

Wywoływanie obiektu

`__call__(self[, args...])`

Metoda `__call__` jest wywoływana gdy wywołujemy nasz obiekt tak jak funkcję. Stąd właśnie definiowane przez nas funkcje są tak na prawdę obiektem typu function, który posiada swoją główna funkcjonalność w metodzie `__call__`.

```
class Foo:
    def init(self, pre, post):
        self.pre = pre
        self.post = post

    def call(self, text):
        print(self.pre + text + self.post)


foo_print = Foo("!!!! ", " ?????")
foo_print("Ala ma kota")
#  !!!! Ala ma kota ?????
```

Kontrola dostępu do obiektu

Metody tej kategorii pozwalają nam na kontrolowanie dostępu do poszczególnych atrybutów naszych obiektów. Przed rozpoczęciem korzystania z poniższych metod watro zaprzyjaźnić się ze specjalnym atrybutem `__dict__`, który zawiera atrybuty obiektu, oraz mieć na uwadze, że niewłaściwe

postepowanie z tymi metodami może doprowadzić do nieskończonej rekurencji.

`__getattr__(self, name)`

Jest to metoda, która jest wywoływana w momencie gdy standardowy dostęp do atrybutu rzuci wyjątek `AttributeError`. Dzięki tej metodzie możesz zwrócić obliczoną wartość atrybutu, który nie istnieje. Jeżeli chcesz ją zaimplementować ale nie zwracać żadnej wartości to rzuć w niej wyżej wspomniany wyjątek.

`__setattr__(self, name, value)`

Wywołanie tej metody następuje w momencie przypisania wartości do atrybutu obiektu. Może się ona przydać gdy np. chcemy zwalidować dane zapisywane do naszego obiektu.

Jeżeli w tej metodzie będziesz zapisywać dane jako atrybut obiektu należy skorzystać z tej metody w klasie bazowej `object`: `object.__setattr__(self, name, value)`

`__delattr__(self, name)`

Podobnie jak w przypadku powyżej metoda ta zastępuje standardowy mechanizm. Implementujemy ją jeżeli chcemy mieć możliwość usuwania zmiennych wewnątrz obiektów w taki sposób: `del obiekt.atrybut`

`__getattribute__(self, name)`

Z tą metodą możesz mieć największe problemy. Jest ona wywoływana zawsze w przypadku dostępu do atrybutu. Przez co łatwo wpaść w nieskończoną pętle jej wywołania. Jeżeli chcesz w jej implementacji uzyskać wartość atrybutu to koniecznie zrób to korzystając z klasy bazowej `object` podobnie jak w przypadku `__setattr__`.

Jednak przed jej implementacją warto zastanowić się czy to `__getattr__` nie jest tą metodą, której potrzebujemy ?

```
class Bar:
    i = {}
    def setattr(self, name, value):
        if value > 0:
            object.setattr(self, name, value)
        else:
            self.i[name] = value
    def getattr(self, name):
        value = self.i[name]
        print("Value from i")
        return value

foo = Bar()
foo.a = 14
foo.b = -14
print(foo.a)
# 14
print(foo.b)
# Value from i
# -14
```

Menadżer kontekstu

O menadżerach kontekstu znajdziesz już jeden artykuł na moim blogu, który dokładnie wyjaśnia to zagadnienie (Jest on tutaj). Tutaj powiemy sobie jedynie o metodach, które zapewniają interfejs menadżera kontekstu i pozwalają na używanie definiowanych przez nas obiektów z wyrażeniem `with`.

`__enter__(self)`

Tutaj definiujemy co ma się stać na początku bloku `with`. Wartość zwrócona z metody `__enter__` zostanie zapisana do zmiennej, której nazwę podajemy po słowie `as`.

`__exit__(self, exc_type, exc_value, traceback)`

W tej metodzie natomiast definiujemy co ma się stać po skończeniu wykonywania kodu z bloku `with` (również gdy zostanie rzucony wyjątek). W tej metodzie zazwyczaj będziemy sprzątać, zamykać zasoby i obsługiwać wyjątki, które wystąpiły. Jako argumenty przyjmuje ona informacje dotyczące wyjątku, jeżeli taki został rzucony. Jeżeli metoda `__exit__` zwraca wartość `True` to jest to informacja dla Pythona, że wyjątki, które wystąpiły w bloku `with` zostały obsłużone.

```
class FileManager():
    def __init__(self, filename, mode):
        print("Metoda __init__")
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        print("Metoda __enter__")
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, type, value, traceback):
        print("Metoda __exit__")
        self.file.close()

with FileManager("log.txt", 'r') as f:
    print("Blok with")
    data = f.read()
```

Obsługa operatorów

Na koniec została już kategoria chyba najprostsza, jednak najbardziej obszerna. Co sprawia, że prawdopodobnie będzie potrzebowała najwięcej czasu na dokładne jej poznanie. Implementowanie metod z tej kategorii pozwala nam nadać specjalnego znaczenia dla operatorów w operacjach ze definiowanymi przez nas typami danych. W innych językach programowania często można spotkać się z pojęciem przeciążania operatorów, gdy wykonujemy operacje analogiczne do tych, o których mowa poniżej.

Operatory porównania

Obsługiwanych w Pythonie operatorów jest na tyle dużo, że możemy podzielić je na kilka kategorii. Pierwszą z nich są operatory porównania. W poniższej tabeli znajdziesz operatory i odpowiadające im metody.

Metoda	Operator
`__lt__`(*self*, *other*)	<
`__le__`(*self*, *other*)	<=
`__eq__`(*self*, *other*)	==
`__ne__`(*self*, *other*)	!=
`__gt__`(*self*, *other*)	>
`__ge__`(*self*, *other*)	>=

Operatory numeryczne

Podobnie sprawa wygląda z operatorami numerycznymi, czyli takimi jak operator dodawania, dzielenia itp. W poniższej tabeli znajdziesz najpopularniejsze z nich.

Metoda	Operator
`__add__`(*self*, *other*)	+
`__sub__`(*self*, *other*)	–
`__mul__`(*self*, *other*)	*
`__truediv__`(*self*, *other*)	/
`__floordiv__`(*self*, *other*)	//
`__mod__`(*self*, *other*)	%
`__pow__`(*self*, *other*)	**

Powyższe operatory implementowane są dla lewego składnika działania. Oznacza to, że wywołując działanie `x + y` zostanie wywołana funkcja: `x.__add__(y)`.

r-operatory

Jeżeli chcemy, żeby zaimplementowane przez nas np. dodawanie było naprzemienne, zaimplementujmy też odpowiednią metodę z kategorii r-operatorów. Może się to przydać zwłaszcza gdy chcemy obsługiwać wbudowane typy danych. Czyli pozwolić zarówno na operację `x + 1` jak i na `1 + x`.

R-operatory tworzymy analogicznie do zwykłych operatorów numerycznych, dodając jedynie literkę „r” na początku. Oznacza to, że metodzie `__add__(self, other)` odpowiada `__radd__(self, other)` itd.

W podobny sposób można obsłużyć operatory przypisania takie jak +=. W tym przypadku dodajemy literkę „i”, Czyli dla wspomnianego operatora += mamy metodę `__iadd__(self, other)`.