POMYSŁ NA EDYTOR

Miałem trochę czasu na zebranie myśli i zastanawiałem się, jak najszybciej by nam się tworzyło kolejne zadania – już wyszliśmy ponad (powszechne w polskim internecie) wpisywanie danych „na sztywno”, bo każde nasze zadanie to uogólniony program, przyjmujący różne wartości. Ale co, gdyby pójść o krok dalej i zautomatyzować też tworzenie tych zadań-programów?

Przede wszystkim wydaje mi się, że w ramach edytora można by uprościć dwie rzeczy – tworzenie zmiennych oraz tworzenie relacji między nimi.

AUTOMATYCZNE TWORZENIE ZMIENNYCH

To rozwinięcie tych uwag, którymi się z Tobą dzieliłem w drodze na skałki (wtedy sam siebie ledwo rozumiałem :p). Jak już wiesz, mój program do regresji (a przynajmniej jego część po stronie serwera) składa się z dwóch modułów:

- części obliczeniowej, oraz
- części zamieniającej na postać tekstową.

Część obliczeniowa „komunikuje się” z częścią zamieniającą na tekst, wysyłając jej pythonowe słowniki (w których kluczem jest nazwa zmiennej, a wartością słownika – wartość zmiennej).

Podczas obliczeń siłą rzeczy powstaje dużo zmiennych pośrednich, ponieważ chcemy pokazać użytkownikom krok po kroku, jak rozwiązywane jest zadanie. Każda ze zmiennych pośrednich musi zatem powtarzać się w obu modułach, przez co trzeba pisać strasznie dużo klocków słownych:

Rys. 1. Ściany tekstu w programie (po lewej: moduł obliczeniowy, po prawej – tekstowy)

```
self.x_transpose = self.x.T
self.xtx = self.x_transpose * self.x
self.xtx_inverted = self.xtx.I
self.xty = self.x_transpose * self.y
self.parameters = self.xtx_inverted * self.xty

self.results = {'x': self.x, 'y': self.y,
        'X_transpose': self.x_transpose,
        'xtx': self.xtx, 'xty': self.xty,
        'xtx inv': self.xtx_inverted,
        'parametry': self.parameters}

return self.results
```

```
# Zamiana elementów słownika na tekst
def create(self):
    x_txt = zamien_macierz(self.wynik_regresji['x'])
    xt_txt = zamien_macierz(self.wynik_regresji['X_transpose'])
    y_txt = zamien_macierz(self.wynik_regresji['y'])
    xtx_txt = zamien_macierz(self.wynik_regresji['xtx'])
    xty_txt = zamien_macierz(self.wynik_regresji['xty'])
    xtx_inv_txt = zamien_macierz(self.wynik_regresji['xtx_inv'])
    parametry_txt = zamien_macierz(self.wynik_regresji['parametry'])
    parametry_opis = wez_wartosc(self.wynik_regresji['parametry'],'','')

    # Fuzja z pozostałym tekstem
    fit_text = [
    r'$$\mathbf{X} = ' + x_txt + '$$',
    r'$$\mathbf{y} = ' + y_txt + '$$',
    r'$$\mathbf{X^{T}X} = ' + xt_txt + x_txt + ' = ' + xtx_txt + '$$',
    r'$$\mathbf{\left(X^TX\right)^{-1}} = ' + xtx_inv_txt + '$$',
    r'$$\mathbf{\hat{\beta}} = \left(X^TX\right)^{-1}X^Ty = '
    + xtx_inv_txt + xty_txt + ' = ' + parametry_txt + '$$'
    ]
```

Czerwonymi ramkami zaznaczyłem wszystkie fragmenty odnoszące się do jednej ze zmiennych – parametrów (wyniku końcowego regresji).

Analizując tę zmienną krok po kroku, mamy cztery etapy – obliczenie wartości, zgranie do słownika, zamianę na tekst i wklejenie w pozostały tekst:

- *self.parameters = self.xtx_inverted * self.xty*
  Tutaj oznacza to mnożenie macierzy, ponieważ numpy pozwala używać operatora *
- *‘parametry’: self.parameters*
  Wpis w słowniku, który zostanie później przejęty przez część tekstową
- *parametry_txt = (...) oraz parametry_opis = (...)*
  Tutaj dwie podobne nazwy, bo najpierw przedstawiam parametry jako całą macierz, a potem opisuję poszczególne elementy (interpretacja parametrów to jedna z najważniejszych rzeczy w zadaniach z regresją)

- *r'$$\mathbf{\hat{\beta}} = (...) + parametry_txt + (...)*
  W tej części postać tekstowa zostaje "wklejona" w pozostały tekst. Początek linijki to tak naprawdę $\hat{\beta}$, czyli popularne oznaczenie na wektor parametrów.

Widzę tu spory potencjał na uogólnienie. Gdyby zrobić z tego coś w rodzaju wpisu w bazie danych, to w najprostszym przypadku mogłoby to wyglądać tak:

Tabela 1. Zadanie z regresją – przykładowy opis zmiennej parameters

Nazwa zmiennej	Rodzaj obiektu	Funkcje	Działanie tworzące			Oznaczenie w Latexu / MathJaxie
			Zmienna tworząca 1	Działanie tworzące	Zmienna tworząca 2	
parameters	macierz	txt, opis	xtx_inverted	[mnożenie macierzy]	xty	\hat{\beta}

A czemu tych pięć pól (z czego jedno trójwartościowe) moim zdaniem wystarczy do pełnego opisania zmiennej? Ponieważ:

1. Jedna nazwa zmiennej, z tego co mi się wydaje, może obsłużyć wiele funkcji w programie. Jedna wartość pola „Nazwa zmiennej" będzie odpowiadała jednocześnie:
   a. nazwie zmiennej w module liczącym
   b. kluczowi słownika
   c. wartości odpowiadającej temu kluczowi
   d. nazwie zmiennej w module tekstowym
2. Pozycja „rodzaj obiektu" pozwoli określić funkcje zamiany na tekst, które można wywoływać na tym obiekcie. Dla obiektu „macierz" będą to:
   a. txt – nazwa zmiennej w module tekstowym będzie równa nazwie z tabeli oraz końcówce _txt; zmienna będzie rezultatem wywołania funkcji zamien_macierz na oryginale (zamiana całości)
   b. opis - nazwa zmiennej w module tekstowym będzie równa nazwie z tabeli oraz końcówce _opis; zmienna będzie rezultatem wywołania funkcji wez_wartosc na oryginale (wyciąganie poszczególnych wartości)
   c. ...I może coś jeszcze, co mi do głowy przyjdzie
3. Pozycja „funkcje" pokazuje, które funkcje zamieniające obiekt na tekst zostaną wykonane w przypadku tej konkretnej zmiennej (w końcu nie trzeba używać wszystkich możliwych); ponieważ dostępne funkcje jasno by wynikały z rodzaju obiektu, wyobrażam to sobie jako coś w rodzaju rozwijanego menu, pokazującego się po wyborze obiektu.
4. Pozycja „działanie tworzące" odpowiadałaby za działanie pokazane w module liczącym (np. *self.parameters = self.xtx_inverted * self.xty);* każdemu takiemu działaniu będą odpowiadać odrębne rzeczy w kodzie programu oraz w LaTeXu; jeśli od jakiejś zmiennej zaczyna się zadanie, to można ją oznaczyć np. jako „pierwotna". Dla powyższego działania:
   a. w kodzie programu '*macierz1 [mnożenie macierzy] macierz2'* będzie odpowiadało ciągowi znaków '*macierz1 * macierz2'* (choć oczywiście mogłoby to być też coś w rodzaju *'matrix.multiply(macierz1, macierz2'))*;
   b. w Latexu dla podstawionych wartości będzie to '*macierz1_txt' + '' + 'macierz2_txt*' (pusty znak, bo mnożenie macierzy zapisuje się przez ustawienie ich obok siebie)
   c. w Latexu dla samych symboli („rozpisywanie wzoru") działanie będzie jak dla powyższego punktu, ale zamiast macierzy weźmie się ich zapisy w Latexu
5. Pozycja „oznaczenie w Latexu" będzie pokazywała symbol, od którego zacznie się linijka wyrenderowana w Latexu. Wydaje mi się, że fajnie by było dać użytkownikom możliwość ustalania własnych symboli (w końcu czasem parametr odpowiadający wyrazowi wolnemu zapisuje się jako α_0, czasem jako β_0 itp.). Dlatego wartość przypisana w tym miejscu przez nas byłaby tylko wartością domyślną.
   Dopuszczam też opcję, żeby w jakimś zadaniu nie zaczynać linijki od symbolu odpowiadającego danej zmiennej; wtedy zamiast tego '\hat{\beta}' można by wpisać 'brak'.

PRAKTYCZNY PRZYKŁAD

Załóżmy, że chcę teraz uogólnić ten powyższy przykład, aby nie trzeba było pisać tego samego dwa razy, w etapie obliczeniowym i tekstowym. W pseudokodzie „szablon” programu z zadaniem mógłby wyglądać tak (nawiasów kwadratowych używam do wskazania zmiennych; reszta to tekst, taki jaki pojawi się w kodzie; działanie tworzące ma wersję *num* w programie liczącym oraz *txt* i *latex* w tekstowym):

```
# Część obliczeniowa

[Zmienna 1] = [DziałanieTworzące_1_num]
[...]
[Zmienna n] = [DziałanieTworzące_n_num]

Słownik = { '[Zmienna 1]': [Zmienna 1], [...], '[Zmienna n]': [Zmienna n] }

# Część tekstowa (na start wczytanie słownika)

[Zmienna_1]_txt = [Funkcja_zmiany_obiektu_na_tekst] (Słownik'[Zmienna_1]')
[...]
[Zmienna_n]_txt = [Funkcja_zmiany_obiektu_na_tekst] (Słownik'[Zmienna_n]')

#Fuzja z pozostałym tekstem

'$$[Oznaczenie_w_Latexu_1] = [DziałanieTworzące_1_latex] =  [Działanie_Tworzące_1_txt]=
[Zmienna_1]_txt$$'
[...]
'$$[Oznaczenie_w_Latexu_n] = [DziałanieTworzące_n_latex] =  [Działanie_Tworzące_1_txt]=
[Zmienna_n]_txt$$'
```

przy czym [DziałanieTworzące_x_latex] oznacza tu, że robię to samo co przy zwykłej tekstowej formie działania, ale zamiast samych wartości obiektów (zamienionych na tekst) biorę ich zapis z Latexa. Odpowiada to sytuacji, gdy najpierw rozpisuję wzór, następnie do niego podstawiam, a na koniec podaję wynik.

Gdyby zastosować taką konwencję jak wcześniej, to 5 zmiennych z pierwszego obrazka (plus dwie zmienne wejściowe) można by zapisać w następujący sposób (kolejność jak w zadaniu):

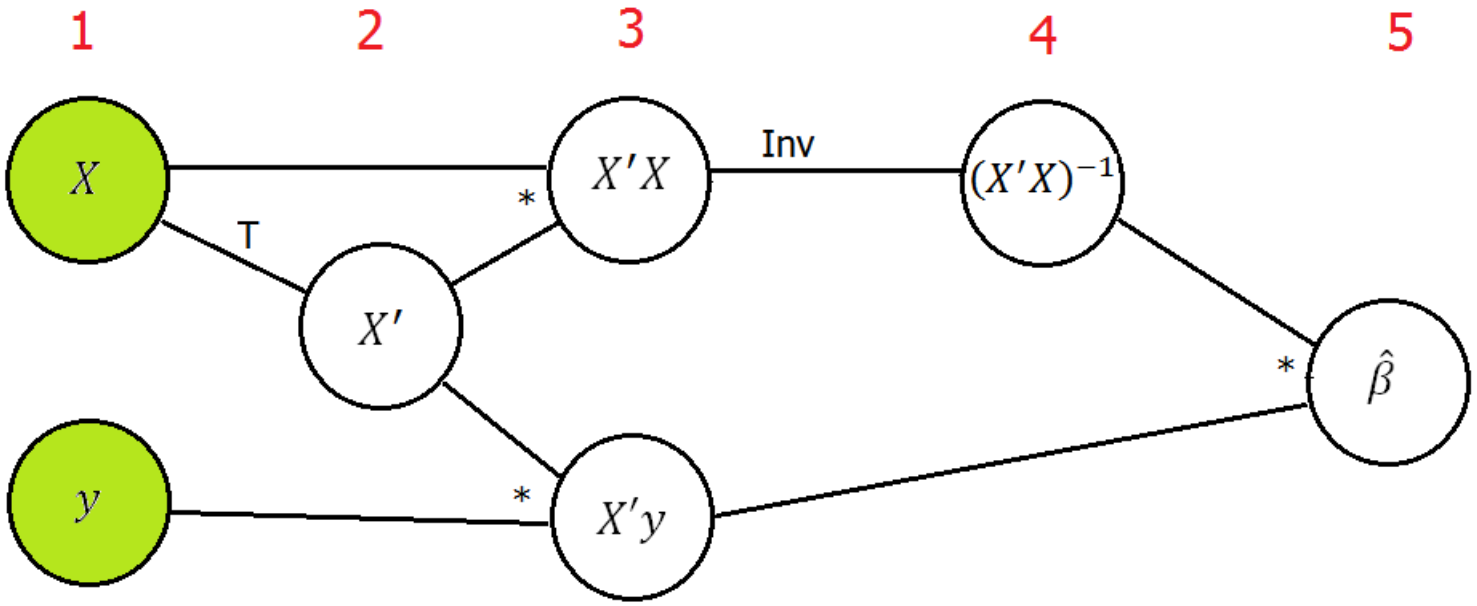
Tabela 2. Zadanie z regresją – pełny opis zmiennych

Nazwa zmiennej	Rodzaj obiektu	Funkcje	Działanie tworzące	Oznaczenie w Latexu / MathJaxie
x	macierz	txt, opis	pierwotna	X
y	macierz	txt, opis	pierwotna	y
x_transpose	macierz	txt, opis	x [transpozycja macierzy]	X’
xtx	macierz	txt, opis	x_transpose [mnożenie macierzy] x	X’X
xtx_inverted	macierz	txt, opis	xtx [odwrócenie macierzy]	(X’X)^{-1}
xty	macierz	txt, opis	x_transpose [mnożenie macierzy] y	X’y
parameters	macierz	txt, opis	xtx_inverted [mnożenie macierzy] xty	\hat{\beta}

W ten sposób samo tworzenie zmiennych wydaje mi się nieco prostsze – ale nadal trzeba je najpierw zdefiniować, a potem wypisać spory blok z kolejnością działania, linijka po linijce. Pomyślałem, że to też dałoby się uprościć.

RELACJE MIĘDZY ZMIENNYMI

Myślałem, myślałem i wymyśliłem, że w sumie kolejność wykonywania działań w zadaniu można przedstawić jako graf – aby przejść do kolejnego węzła, trzeba zaliczyć poprzednie. Tutaj wersja dla regresji z poprzedniego zadania:



Przez * oznaczyłem działanie mnożenia macierzy, T to transpozycja, Inv to odwrócenie. Zielonym kolorem oznaczyłem zmienne pierwotne.

Liczba zmiennych równa się liczbie wierzchołków grafu – dla każdej z nich trzeba by stworzyć osobny „rekord” przy definiowaniu programu-zadania.
Cyfry u góry odpowiadają kolejnym krokom rozwiązywania zadania. Na przykład w kroku 3 nie ma znaczenia, czy najpierw pomnożysz przez siebie X' i X, czy też X' i y; te węzły są równorzędne. Nie będziesz natomiast w stanie wykonać żadnego z tych działań przed wykonaniem samej transpozycji.

Gdyby w jakiś sposób udało nam się zaimplementować projektowanie zadań z użyciem grafu, to widzę tutaj następujące korzyści:

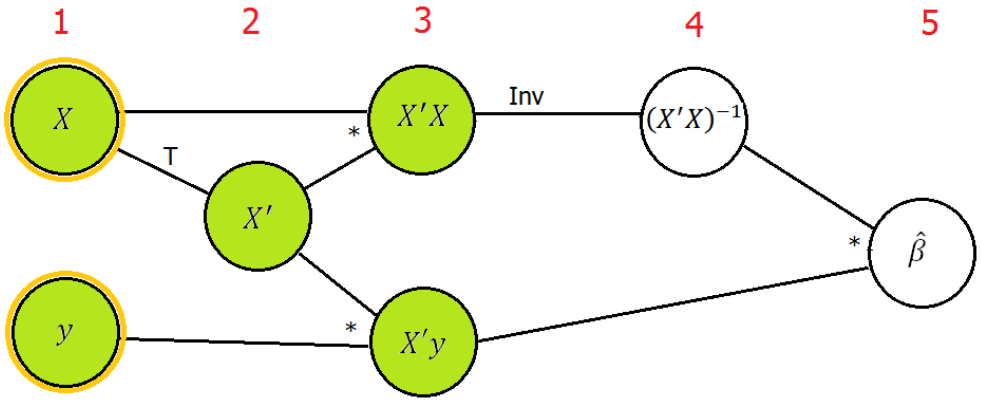
Dla nas:

- Możliwość tworzenia ciekawszych zadań – dzięki grafowi widać np., że tak naprawdę do rozwiązania zadania wystarczą zmienne X' oraz y. Nawet jeśli nie masz samego iksa, to „odzyskujesz go”, wykonując transpozycję. W ten sposób możemy zaczynać od różnych punktów, tworzyć różnorodne zadania dla jednego zagadnienia.
- Tworzenie takich zadań nie musiałoby się wiązać z dodatkową pracą – po prostu wpisujemy jakąś zmienną jako „pierwotną”, podajemy jej typ, a nasz edytor podsuwa użytkownikowi odpowiedni interfejs (osobny dla macierzy, zmiennych logicznych itp.). Podajemy pozostałe wartości, a on już się zatroszczy o wykonanie odpowiedniego rodzaju obliczeń, zamiany na tekst itp. Może nawet dałoby się tworzyć automatyczne testy, określając warunki skrajne dla każdego obiektu (np. wyznacznik równy zero dla macierzy)?
- Może to być dla nas dodatkową kontrolą poprawności przy tworzeniu zadań – ciężko mi określić ścisłą regułę, ale powiedziałbym coś takiego: „Od punktów startowych przesuńcie się w tył z użyciem odwracalnych operacji na jednej zmiennej (takich jak transpozycja czy odwracanie). Jeśli dalej brakuje jakiejś zmiennej „do pary” przy operacji dla dwojga (np. mnożeniu macierzy), to coś jest sknocone i nie da się użyć takiej kombinacji punktów startowych w zadaniu”.
  Przy takim sprawdzaniu wyszłoby np. na jaw, że znajomość $X'X$ oraz y to za mało do obliczenia parametrów beta (chyba że nie znam jakiejś metody), natomiast X' oraz y już są wystarczające
- Inny komunikat ostrzegawczy – gdyby powstał nam luźny wierzchołek, „niepowiązany” żadnymi krawędziami z pozostałymi wierzchołkami; takie coś nijak nie pomoże w rozwiązaniu zadania. Choć w sumie mógłby trafić do zadania „dla zmyły”

Dla użytkowników:

- Dodatkowa atrakcja i interaktywność; gdybyśmy udostępniali graf zadania użytkownikom, to w fazie nauki ktoś może na żywo oglądać swoje postępy w zadaniu i zyskiwać wskazówki a propos dalszego działania (np.

jest w kroku 3 zadania z regresją, więc teraz wypadałoby odwrócić $X'X$). Coś w tym stylu:



- Standaryzacja – tak jak wspomniałem, określanie przez nas rodzaju obiektów początkowych pozwoliłoby automatycznie tworzyć dla nich interfejs. Dzięki temu użytkownicy mieliby pewną spójność – np. za każdym razem, kiedy trzeba by wprowadzić macierz, ich oczom ukazywałby się dokładnie ten sam interfejs.
- Brak strat w szybkości działania – edytor służyłby tylko do tworzenia programów (plików tekstowych) po naszej stronie, zatem nic by się nie zmieniało po stronie użytkownika. Co najwyżej zyskałby nowe zadania do odwiedzenia, jeśli włączymy nowo utworzony program do backendu i podepniemy pod jakiegoś linka na stronie.

To teraz pytanie – jak sobie wyobrażam wprowadzanie takich zmiennych w Pythona? W sumie to ciężkie pytanie, bo ciężko mi pisać zwykłe programy, a co dopiero taki „metaprogram” :D ale mam parę propozycji:

Gdzieś tam powinna się znaleźć funkcja *cośtam(*args)*, bo podobno gwiazdka pozwala wsadzić dowolną ilość argumentów. Załóżmy, że „karmiłoby się” ją listami 7-elementowymi (liczba kategorii określających zmienne proponowana przeze mnie powyżej – „obiekt”, „zapis w Latexu” itp.).
Drugi warunek: taka funkcja musiałaby tworzyć programy Pythona – zatem domyślam się, że powinna tworzyć trzy/cztery pliki tekstowe z rozszerzeniem .py (obliczenia, zamianę na tekst, interfejs oraz ew. test).

Moglibyśmy mieć do tej funkcji coś w rodzaju metainterfejsu – tak jak wcześniej wartości do pojedynczego wektora, tak teraz wprowadzałoby się poszczególne wektory (i macierze, i inne rzeczy) występujące w danym rodzaju zadania.

Załóżmy więc, że wprowadziłem przez interfejs poprzednie zmienne, jak w tabeli 2, i że teraz chcę je ze sobą powiązać. Powiedzmy, że funkcję *chain* stosowałoby się do narzucenia prostej kolejności działań, jedno po drugim. Wpisałbym (na wałkowanym wcześniej przykładzie):

```
chain(x, y, x_transpose, xtx, xtx_inverted, xty, parameters.1, parameters.2)
```

...a funkcja już by sobie ustaliła na podstawie elementów list, co może być poprzednikiem, a co następnikiem i narysowała ładny graf :) te parameters.1 i .2 wyobrażam sobie jako metody zamiany na tekst (żeby funkcja „spojrzała”: lista parameters, pozycja 2 -> obiekt „macierz” -> ma dwie metody zamiany na tekst -> oho, użyję pierwszej z nich -> oho, a teraz drugiej);
to oczywiście nie muszą być metody klasy, tak tylko dałem w przykładzie. Grunt, żeby parameters było liczone tylko raz, a jedynie zamienione na tekst na dwa sposoby, w kolejności podanej przez użytkownika. Ogólnie: żeby funkcja chain przyjmowała dowolną liczbę argumentów, ale uwzględniała tylko te unikalne, niepowtarzające się.

Oprócz funkcji *chain* powinna też być funkcja umożliwiająca wcześniejsze przerwanie zadania, jeśli nie chcemy robić całego; tak jak w przypadku regresji mogliśmy skończyć po obliczeniu parametrów albo po ustaleniu odchyleń. Dajmy na to coś takiego:

```
stop_if_flag_set(tekst="")
```

Taka funkcja powodowałaby kilka rzeczy:

- do pliku interfejs.py dla danego zadania dodawałaby możliwość ustawienia wartości flagi boolowskiej (wyświetlając przy tym tekst wrzucony jako argument, np. „Czy chcesz policzyć błędy oszacowania parametrów?” – pytanie zawsze powinno się odnosić do bloku następującego PO nim). Ponadto, skoro

interfejs będzie tworzył pliki Jsona i wysyłał je na serwer, powinny one mieć dodatkowe miejsce na każdą z flag;
- do pliku obliczeniowego dodawałaby instrukcję warunkową – jeśli flaga z pliku Jsona będzie ustawiona na False, to return zwraca jedynie słownik znajdujący się przed tą instrukcją i zawierający wyniki poprzednich obliczeń, a także samą flagę.
- analogicznie w pliku z zamianą na tekst; myślę, że nie ma co przeszukiwać słownika pod kątem nieistniejącego klucza, więc funkcja może po prostu odczytać ew. flagę False z modułu liczącego i też zwrócić tylko słownik z tym, co miała zamienić na tekst

Wyobrażam sobie na przykład możliwość stworzenia następującego obiektu (dla większych zadań):

```
workflow =  [chain(x, y, x_transpose, xtx, xtx_inverted, xty, parameters.1, parameters.2),
             stop_if_flag_set(tekst=" Czy chcesz policzyć błędy oszacowania parametrów?"),
             chain(...)]
```

oczywiście musiałbym pomyśleć nad sposobem, w jaki drugi chain mógłby kontynuować od pierwszego... może lepiej byłoby mieć wszystkie stop_if_flag_set jako argumenty do jednej funkcji, wstawiane tam, gdzie chcemy żeby zaszły? Ale nad tym będę jeszcze musiał dużo pomyśleć.

Niezależnie od technikaliów załóżmy, że mam już „gotowca" – chainy i ewentualne punkty przerwania ustawione, graf się zgadza, zadanie jest gotowe do stworzenia. Teraz odpalam funkcję, która przyjmuje ten zbiorczy obiekt:

```
create_exercise(workflow)
```

I wyobrażam sobie, że zrobi ona szereg rzeczy:

- Stworzy plik interfejs.py zawierający miejsce na wprowadzenie zmiennych wskazanych jako pierwotne dla zadania (np. wektora y i macierzy X), miejsce na ustawienie flag mówiących, jaką część zadania chcemy zrobić, a także miejsce na konfigurację domyślnego zapisu zmiennych w danym zadaniu – jeśli np. chcemy zamiast β wyświetlać α).
- Stworzy plik obliczenia.py, w którym po kolei, we właściwej kolejności, zostaną wgrane równania poszczególnych zmiennych. We wskazanych miejscach zostaną umieszczone punkty przerwania, aktywowane w razie ustawienia flag na False poprzez interfejs. Na końcu funkcja będzie eksportowała słownik o postaci: {'zmienna': zmienna, ...}
- Stworzy plik wersja_tekstowa.py, w którym zmienne z poprzedniego pliku będą zamieniane na tekst. Liczba zmiennych „tekstowych" nie musi być równa liczbie zmiennych obliczonych (można np. ustalić, żeby najpierw renderowana była cała macierz, a potem jej poszczególne elementy – to już dwa warianty na jeden obiekt).
- Stworzy plik test.py (lub zbiorczy.py), który będzie zawierał funkcje pozwalające odpalić pozostałe moduły dla nietypowych wartości zmiennych pierwotnych/wejściowych (np. macierzy o wyznaczniku równym zero; liczb równych zero itp.; o ich wyborze będzie decydowała kombinacja dwóch zmiennych z tabeli – „obiekt" oraz wartość „pierwotna").

Jeśli spojrzysz na blok kodu zaraz pod nagłówkiem Praktyczny Przykład, to tę całą funkcję z edytora wyobrażam sobie trochę jak takie list comprehension rozwijające się w te bloki kodu, które powtarzają się w obu modułach :)

SPECJALNE PRZYPADKI

Mam świadomość, że taki edytor nie ogarnie wszystkich skrajnych przypadków. Np. już sama regresja ma tę opcję wstawiania wyrazu wolnego, która wymaga zarówno dodatkowej flagi w interfejsie, jak też dodatkowego działania w części obliczeniowej. Jednak w takim przypadku można po prostu dodać regułę ręcznie do programu, w oparciu o stworzony „szkielet".
Wydaje mi się, że możliwość tworzenia zadań przez edytor pomoże szczególnie w przypadku powtarzających się zamian obiektów na tekst – każda całka będzie miała swoje niezmienne metody konwersji, każda macierz – swoje.

Daj znać, co o tym wszystkim myślisz :)

MOTYWACYJNY PRZYKŁAD

Na koniec dorzucę jeszcze link do artykułu, w którym facet wspomina o przewadze konkurencyjnej swojego startupa – polegała ona na tym, że byli w stanie bardzo szybko wdrażać nowe funkcjonalności. Kiedy tylko konkurencja czymś się chwaliła, brali się do roboty i wypuszczali to przed nimi.

```
What were the results of this experiment? Somewhat surprisingly, it worked. We eventually had
many competitors, on the order of twenty to thirty of them, but none of their software could
compete with ours. We had a wysiwyg online store builder that ran on the server and yet felt
like a desktop application. Our competitors had cgi scripts. And we were always far ahead of
them in features. Sometimes, in desperation, competitors would try to introduce features that
we didn't have. But with Lisp our development cycle was so fast that we could sometimes
duplicate a new feature within a day or two of a competitor announcing it in a press release.
By the time journalists covering the press release got round to calling us, we would have the
new feature too.
```

Działali wprawdzie w oparciu o Lispa, nie Pythona, ale podobno sprowadzało się to właśnie do możliwości działania na takich „metaprogramach”. Z tego co kojarzę, twórcy gier też najwięcej czasu poświęcają na stworzenie spójnego edytora plansz, a potem już w nim szybko robią resztę :) więc może coś w tym jest.

Tu link: http://paulgraham.com/avg.html