

Zestaw 4

1. Zadanie ma na celu sprawdzenie jak wygląda tworzenie obiektów dla typów z wielokrotnym dziedziczeniem, jakie funkcje `__new__` oraz `__init__` są lub nie są wywoływane. Wychodzimy od dwóch klas bazowych (identycznych, różnią się tylko nazwą), `class Baza(object)` oraz `class A(object)` – patrz plik [zadanie1.py](#). Posiadają one napisane `__new__`, `__init__`, `__str__` oraz funkcję `id()`. Proszę przestudiować kilka różnych wariantów klas potomnych (B, C, D...) oraz tworzenia odpowiednich obiektów. Proponowane scenariusze są zapisane w pliku, klasy potomne powinny mieć zawartość zbliżoną, w celach studyjnych, do klas bazowych. W programie uruchomieniowym prezentować (oraz potrafić przedyskutować co się dzieje) różne scenariusze, włączając zagadnienie MRO (Method Resolution Order).
2. W pliku [zadanie2.py](#) (załączonym) znajduje się szkielet kodu klasy `Zespolona`. Posiada ona dwie składowe instancje, `r` (real) oraz `i` (imaginary), odpowiadające części rzeczywistej i urojonej liczby zespolonej. Dodatkowo zdefiniowane są funkcje sprzężenia zespolonego (`conjugate`) oraz fazy (`argz`). Reszta to szereg metod specjalnych `__NNN__`, których treść (w miejsce `pass`) należy napisać tak, żeby nastąpiło poprawne wykonanie kodu w funkcji `main()` – oczekiwane wyniki zapisano w komentarzu. Jeśli trzeba, to w internecie poszukać informacji o liczbach zespolonych. Proszę zwrócić uwagę, że typ `float` czy `int` wymaga specjalnej obsługi w kodzie.
3. Zadanie przeglądowe, w ramach którego należy napisać proste ilustrujące fragmenty kodu. Należy kierować się wskazówkami z pliku [zadanie3.py](#) tak, żeby:
 - a) wyjaśnić zasadę funkcji instancji klasy, funkcji składowej klasy z użyciem dekoratora `@classmethod`, funkcji statycznej klasy z użyciem dekoratora `@staticmethod`
 - b) klasę abstrakcyjną, dziedziczącą z klasy `ABC` oraz metodę abstrakcyjną z użyciem dekoratora `@abstractmethod`, wraz z klasami potomnymi i odpowiednimi implementacjami
 - c) przykład atrybutu klasy definiowanego z pomocą dekoratora `@property` oraz odpowiedniego `@nazwa.setter`
4. Dynamiczny charakter języka Python nie pozwala na bezpośrednie przeładowywanie funkcji o tych samych nazwach, ale różnych argumentach. Z pomocą dekoratorów pojawiają się w języku techniki emulujące takie zachowania. W ramach zadania proszę przestudiować materiał na temat `singledispatch` oraz `singledispatchmethod` z modułu `functools` oraz napisać dowolny kod ilustrujący te przypadki (inny niż w podanej dokumentacji) ➡ <https://docs.python.org/3/library/functools.html#functools.singledispatch>
5. Dyskutowane w zadaniu poprzednim rozwiązania mają pewne ograniczenia, jest to nowa funkcjonalność w języku Python, która nadal nie obejmuje bardziej skomplikowanych zastosowań (np. przypadków dziedziczenia). W tym zadaniu przyjrzymy się zewnętrznemu modułowi `multipledispatch` (prawdopodobnie trzeba go najpierw zainstalować: `pip install multipledispatch`):
 ➡ <https://github.com/mrocklin/multipledispatch/>
 Przykład w pliku [zadanie5.py](#) to klasy `Figura`, `Prostokat`, `Kwadrat` i chodzi o to, aby definiując poniżej wersje funkcji `pole`, różniące się argumentami oraz liczbą argumentów, wywołania wersji funkcji `pole` było uzależnione właśnie od argumentów. Jeśli ta selekcja ma się odbywać na więcej niż jednym argumencie, można właśnie użyć `multipledispatch` oraz dekorator `@dispatch`. Należy dopisać brakujący kod, aby testowe przykłady działały. P.s. jeszcze inna opcja, to zewnętrzny moduł `plum`
 ➡ <https://wesselb.github.io/plum/intro.html>