

Temat 1: Python

WPROWADZENIE

- Zanim zaczniemy używać Pythona do wykonywania zadań przewidzianych programem kształcenia dla przedmiotu, potrzebujemy przynajmniej pobieżnej wiedzy na temat sposobu jego działania.
- Ta część zajęć nie ma na celu uczynienie z Was ekspertów od Pythona, jednak ma zapewnić wystarczającą ilość informacji, aby umożliwić zrozumienie przykładowego kodu wraz z komentarzami.
- W poszczególnych rozdziałach zostały zamieszczone najważniejsze elementy języka w materiale pozwalającym zrozumieć jak Python wykonuje różne zadania.
- Na przykład, aby umieć określić, co robi przykładowy kod z danymi, trzeba wiedzieć, jak Python przekształca różne rodzaje danych.
- Podstawowe informacje na temat wykorzystania danych liczbowych, logicznych, tekstowych i dat znajdują się w pierwszej części skryptu.

Cel zajęć

Celem tych zajęć jest zaprezentowanie wybranych podstawowych elementów języka Python oraz zbioru bibliotek przeznaczonych do przetwarzania danych, dzięki którym będzie można skutecznie i wszechstronnie dokonywać analizowania danych lub przetwarzać obrazy.

Lista zagadnień do nauki w ramach danego tematu:

1	Podstawowe informacje o języku Python i środowisku programowym Anaconda	4
1.1	Coś o Pythonie i jego wykorzystaniu do przetwarzania danych.....	4
1.2	Dwa polecane środowiska Pythona wykorzystywane do obliczeń naukowych, uczenia maszynowego, analizy danych i przetwarzania obrazów:.....	5
2	Instalacja Pythona – środowisko Anaconda	6
3	Krótkie wprowadzenie do środowisk Conda, Navigator, Jupyter i Spyder.....	7
3.1	Conda.....	7
3.2	Anaconda Navigator	7
3.2.1	Uwagi ogólne:.....	7
3.2.2	Zarządzanie środowiskami:.....	8
3.2.3	Zarządzanie Pythonem:	8
3.2.4	Zarządzanie pakietami:.....	8
3.3	Jupyter Notebook	9
3.3.1	Ogólny opis środowiska <i>Jupyter Notebook</i>	9
3.3.2	Określenie repozytorium dla przedmiotu (jako przykład).....	10
3.3.3	Definiowanie folderu roboczego dla przedmiotu (jako przykład)	10
3.4	Spyder	11
3.5	Google Colab.....	11
3.6	Krótkie ćwiczenia wprowadzające do prezentowanych narzędzi	11
4	Przegląd komponentów języka Python	15
4.1	Podstawy języka	15
4.1.1	Typy danych języka Python.....	15
4.1.2	Zmienne i ich konwencje w Pythonie	26
4.1.3	Instrukcja print()	27
4.1.4	Instrukcja input()	32
4.1.5	Operator wycinania <i>[start:stop:step]</i>	32
4.2	Moduły wbudowane.....	35
4.3	Struktury danych: listy, krotki, słowniki, zbiory.....	36
4.3.1	Listy.....	36
4.3.2	Krotki	39
4.3.3	Słowniki.....	41
4.3.4	Zbiory.....	43
4.4	Kontrola przepływu programu	45
4.4.1	Instrukcja warunkowa if	46
4.4.2	Pętla for	48

4.4.3	Pętla while	54
4.5	Funkcje.....	55
4.5.1	Przegląd wbudowanych funkcji Pythona.....	55
4.5.2	Definiowanie własnych funkcji	57
5	Przykładowe zbiory danych	60
5.1	Przykładowe zbiory danych dostępne w ogólnodostępnych zasobach oraz w pakiecie <code>scikit-learn</code>	60
5.1.1	Zbiór danych <i>Iris</i>	60

1 Podstawowe informacje o języku Python i środowisku programowym Anaconda

1.1 Coś o Pythonie i jego wykorzystaniu do przetwarzania danych

1. Python pojawił się w 1991.
2. Jest popularnym interpretowanym językiem programowania.
3. Wokół Pythona rozwinęła się duża społeczność użytkowników zajmujących się obliczeniami naukowymi i analizą danych (nawet bardzo dużych), uczenia maszynowego, tworzenia oprogramowania naukowego i przemysłowego.
4. Zastosowanie Pythona w analizie danych, obliczeniach interaktywnych i wizualizacji danych można porównać z innymi otwartymi i komercyjnymi językami i narzędziami, takimi jak *Matlab*, język *R*, *SAS*.
5. W ostatnich latach Python umocnił się w obszarze budowania oprogramowania ogólnego przeznaczenia, a zestaw różnorodnych bibliotek, które zostały przygotowane do wspierania Pythona sprawiają, że nadaje się on doskonale do roli głównego języka używanego do budowania aplikacji przetwarzających dane.
6. Sukces Pythona w obszarze obliczeń naukowych ma swoje przyczyny w tym, że łatwo się integruje z kodem C, C++, FORTRAN, a w tych językach przez lata społeczność naukowa zgromadziła olbrzymie zestawy bibliotek przeznaczonych do algebry liniowej, optymalizacji, integracji, szybkich transformacji Fouriera oraz innymi tego typu obliczeniami.
7. Bardzo często w środowiskach produkcyjnych używa się tzw. "dwujęzyczności", tzn. w praktyce opracowywanie, prototypowanie i testowanie nowych pomysłów przeprowadza się w bardziej wyspecjalizowanych do tego językach programowania, np. R czy SAS, a następnie przeniesienie gotowego i sprawdzonego pomysłu do większego systemu produkcyjnego napisanego w jednym z języków: Java, C++ lub C#. Jednak coraz częściej Python wypełnia oba te podejścia, tzn. zastępuje utrzymywanie dwóch środowisk programistycznych jednym. Dzięki temu zarówno badacze jak i programiści posługują się tym samym zestawem narzędzi.

1.2 Dwa polecane środowiska Pythona wykorzystywane do obliczeń naukowych, uczenia maszynowego, analizy danych i przetwarzania obrazów:

1. Podstawowa dystrybucja środowiska programistycznego jest dostępna na wszystkie systemy operacyjne pod adresem: <https://www.python.org/>
2. Bardzo polecaną dystrybucją Pythona przeznaczoną do obliczeń naukowych jest **Anaconda** stworzona przez firmę *Continuum Analytics*.
3. **Anaconda** jest bezpłatną dystrybucją również w przypadku zastosowań komercyjnych i zawiera wszystkie niezbędne pakiety wykorzystywane w analizie danych, obliczeniach matematycznych oraz inżynierii.
4. W witrynie projektu napisano, że **Anaconda** ma stanowić gotową do zastosowań przemysłowych dystrybucję Pythona przeznaczoną do przetwarzania danych, analityki predykcyjnej i obliczeń naukowych na dużą skalę.

1. Opis środowiska **Anaconda Individual Edition**

1. **Anaconda Individual Edition** zawiera programy **conda** i **Anaconda Navigator** oraz interpreter Pythona wraz z setkami naukowych pakietów, a wszystko to instaluje się razem podczas jednego procesu instalacji.
2. **Conda** to menedżer pakietów, który działa w interfejsie wiersza poleceń, takim jak **Anaconda Prompt** w systemie Windows.
3. **Navigator** to graficzny interfejs użytkownika, który umożliwia uruchamianie aplikacji i łatwe zarządzanie pakietami **conda**, środowiskami i kanałami bez użycia poleceń wiersza poleceń.
4. Oba narzędzia (**conda** i **Navigator**) są wzajemnie zamienne i można część działań wykonywać w jednym, a część w drugim w zależności od własnych preferencji.

2 Instalacja Pythona – środowisko Anaconda

1. Instalację środowiska **Anaconda Individual Edition** należy przeprowadzić ściągnając instalator ze strony:

<https://docs.anaconda.com/anaconda/install>

i wykonując zawarte polecenia podczas procesu instalacji.

2. Po zainstalowaniu **Anacondy** można instalować nowe pakiety Pythona za pomocą polecenia:

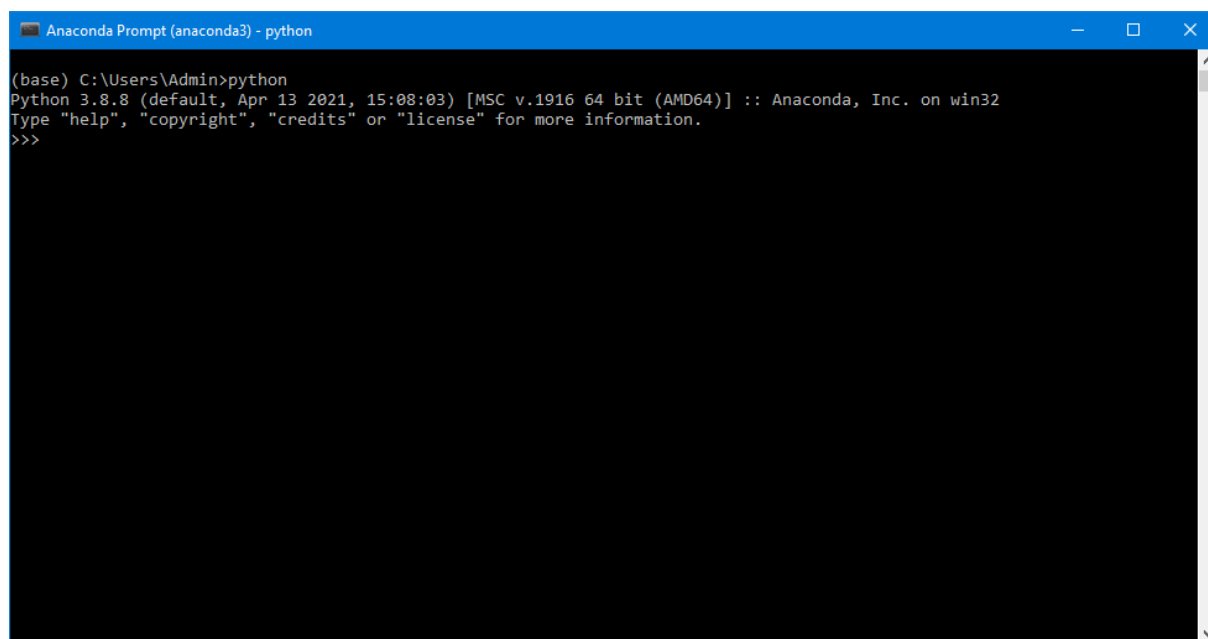
```
conda install JakiśPakiet
```

3. Zainstalowane pakiety można zaktualizować poleceniem:

```
conda update JakiśPakiet
```

4. Uruchomienie Pythona z menu **Start -> Anaconda3 -> Anaconda Prompt (anaconda3)** i wprowadzenie polecenia **Python**

5. Powinno pojawić się okno podobne do poniższego



```

Anaconda Prompt (anaconda3) - python
(base) C:\Users\Admin>python
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

3 Krótkie wprowadzenie do środowisk Conda, Navigator, Jupyter i Spyder

3.1 Conda

1. Zarządzanie pakietami, zależnościami i środowiskiem dla dowolnego języka — Python, R, Ruby, Lua, Scala, Java, JavaScript, C/C++, FORTRAN i nie tylko.
2. Program wiersza poleceń **Conda** to system zarządzania pakietami typu open source i system zarządzania środowiskiem, który działa w systemach Windows, macOS i Linux.
3. **Conda** jest zarówno menedżerem pakietów, jak i menedżerem środowisk. Pomaga to analitykom danych zapewnić, że każda wersja każdego pakietu ma wszystkie wymagane zależności i działa poprawnie.
4. **Conda** szybko instaluje, uruchamia i aktualizuje pakiety oraz ich zależności.
5. Wiele naukowych pakietów zależy od konkretnych wersji innych pakietów. Analitycy danych często używają wielu wersji wielu różnych pakietów i używają wielu środowisk do wydzielenia tych różnych wersji.
6. **Conda** z łatwością tworzy, zapisuje, ładuje i przełącza się między środowiskami na lokalnym komputerze.
7. Program został stworzony dla programów Pythona, ale może pakować i dystrybuować oprogramowanie dla dowolnego języka.
8. **Conda** jako menedżer pakietów pomaga znaleźć i zainstalować pakiety.
9. Jeśli potrzebny jest pakiet, który wymaga innej wersji Pythona, nie ma potrzeby przełączać się do innego menedżera środowiska, ponieważ **conda** jest również menedżerem środowiska.
10. Za pomocą kilku poleceń można skonfigurować całkowicie oddzielne środowisko do uruchamiania innej wersji Pythona, jednocześnie kontynuując uruchamianie zwykłej wersji Pythona w normalnym środowisku.
11. W swojej domyślnej konfiguracji **conda** może zainstalować i zarządzać tysiącami pakietów na **repo.anaconda.com**, które są tworzone, przeglądane i utrzymywane przez **Anaconda®**.

3.2 Anaconda Navigator

3.2.1 Uwagi ogólne:

1. **Anaconda Navigator** jest graficznym interfejsem użytkownika dla menadżera pakietów i środowiska, programu **conda**.
2. **Navigator** to prosty sposób pracy z pakietami i środowiskami, oparty na interfejsie graficznym, bez konieczności wpisywania poleceń **conda** w oknie terminala. Można go użyć, aby znaleźć

żądane pakiety, zainstalować je w środowisku, uruchomić pakiety i zaktualizować je – wszystko w programie **Navigator**.

3. Uruchomienie:

 menu **Start** -> **Anaconda Navigator**

 menu **Start** -> **Anaconda Prompt** i polecenie **anaconda-navigator**

4. Podczas uruchomienia **Navigator** sprawdza, czy jest zainstalowana **Anaconda**.


5. Jeżeli **Navigator** nie wystartuje należy sprawdzić poprawność instalacji **Anacondy**.

6. Podczas uruchamiania **Navigator** sprawdza czy jest dostępna nowa wersja. Jeśli jest nowsza wersja od zainstalowanej należy dokonać aktualizacji.

3.2.2 Zarządzanie środowiskami:

1. **Navigator** używa **conda** do tworzenia oddzielnych środowisk zawierających pliki, pakiety i ich zależności, które nie będą wchodzić w interakcje z innymi środowiskami.

2. Przykład utworzenia nowego środowiska:

 W **Navigatorze** kliknij kartę **Environments** i kliknij przycisk **Create**, zostanie otwarte okno **Create new environment**.

 W polu **name** napisz nazwę dla Twojego środowiska, np. **test_env** i kliknij **Create**.

 **Navigator** utworzy nowe środowisko i aktywuje je.

 Teraz istnieją dwa środowiska: **base (root)** i **test_env**.

 Można przełączać się pomiędzy środowiskami.

3.2.3 Zarządzanie Pythonem:

1. Kiedy jest tworzone nowe środowisko **Navigator** instaluje te same wersje pakietów, które instalowały **Anacondę**. Jeżeli jest potrzeba użycia innej wersji Pythona w tworzonym środowisku należy podczas jego tworzenia zaznaczyć wersję właściwą.

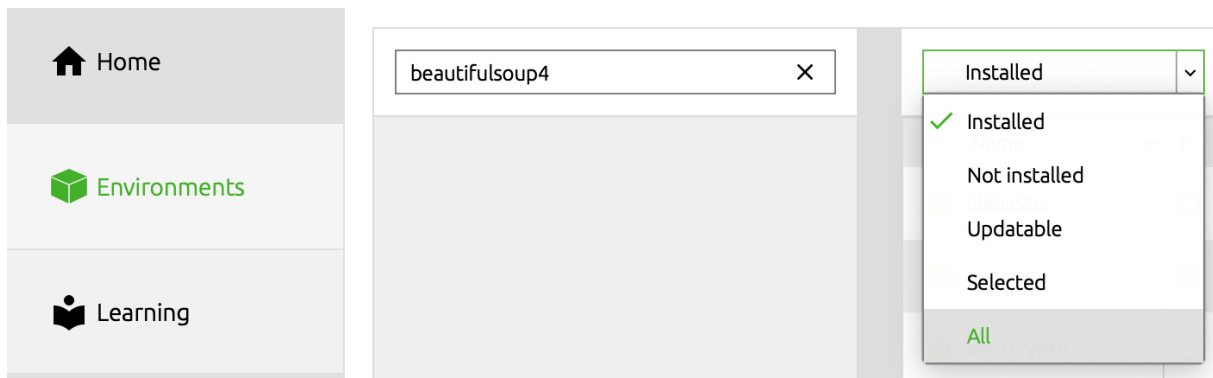
3.2.4 Zarządzanie pakietami:

1. Sprawdzanie które pakiety są zainstalowane, które są dostępne oraz wyspecyfikowanie pakietów i ich instalacja.

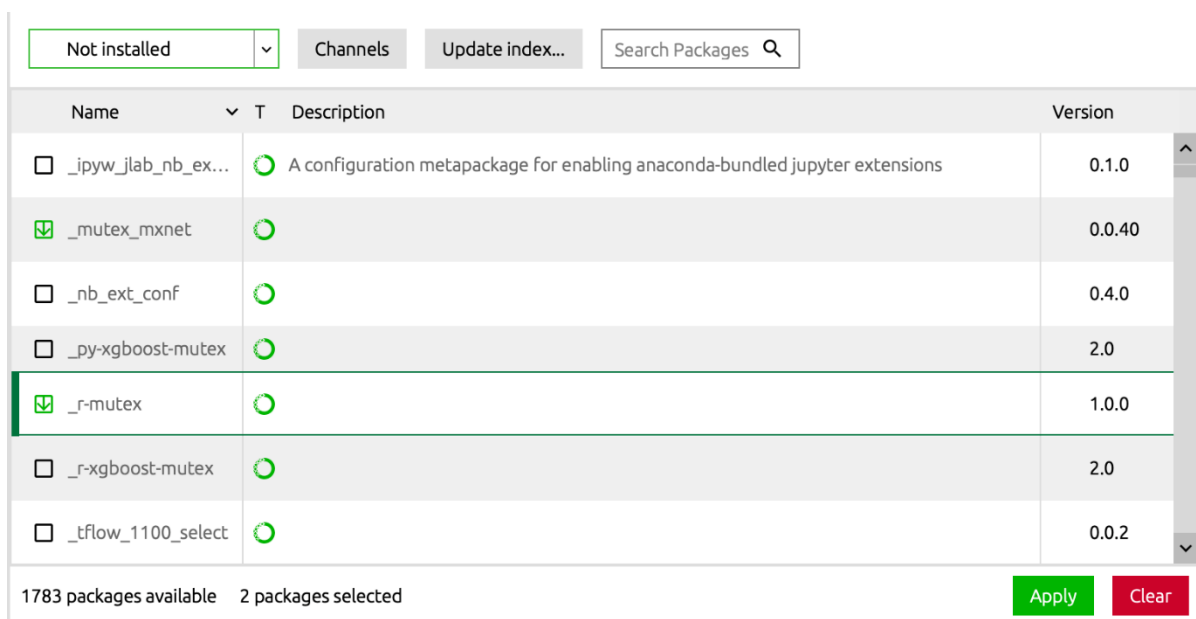
2. Aby znaleźć pakiet który już był instalowany kliknij nazwę środowiska które chcesz przeszukiwać. Zainstalowane pakiety wyświetlają się w prawym panelu.

3. Wybór wyświetlanych pakietów można zmienić wybierając odpowiednią opcję w polu wybieralnym: **Installed**, **Not installed**, **Updatable**, **Selected**, **All**.

4. Sprawdź czy pakiet którego nie zainstalowałeś o nazwie **beautifulsoup4** jest dostępny w repozytorium **Anacondy** (musisz być podłączony do Internetu). Będąc na karcie **Environments**, w polu wyszukiwania **Search Packages** napisz **beautifulsoup4**, a z **Search Subset** wybierz **All** lub **Not Installed**. Coś tak, jak poniżej:







5. Chcąc zainstalować jakiś pakiet w danym środowisku, zaznacz pole wyboru obok nazwy danego pakietu i kliknij przycisk **Apply**.



3.3 Jupyter Notebook

3.3.1 Ogólny opis środowiska Jupyter Notebook

1. Interfejs pozwala łatwo tworzyć pliki notatników z kodem w Pythonie zawierające dowolną liczbę przykładów, z których każdy może działać indywidualnie.
2. Kod działa w przeglądarce, więc nie ma znaczenia jakiej platformy używa się do pracy.
3. Głównym zadaniem aplikacji **Jupyter Notebook** jest ułatwienie opowiadania, niezbędnego w nauce o danych, uczeniu maszynowym i naukach powiązanych, ponieważ zwykle w tych obszarach potrzebne są następujące możliwości:

-  wyświetlanie pośrednich (diagnostycznych) wyników z każdego etapu rozwijanego algorytmu;
-  uruchamianie tylko wybranych sekcji (komórek) kodu;
-  zapisywanie pośrednich wyników i możliwości kontroli ich wersji;
-  prezentowanie pracy za pomocą połączenia tekstu, kodu i grafiki.

Wszystkie powyższe cechy posiada **Jupyter Notebook**.

4. Po zapisaniu notatnika **Jupytera** otrzymuje się plik **.ipynb** w formacie **JSON**. Zawiera on wszystkie komórki i ich zawartości oraz dane wyjściowe. Ułatwia to pracę, ponieważ nie trzeba uruchamiać kodu by zapoznać się z notatnikiem. Jest to bardzo wygodne, zwłaszcza gdy dane wyjściowe obejmują rysunki, a kod wymaga wykonania czasochłonnych procedur.
5. Wadą notatników **Jupyter** jest to, że format pliku **JSON** nie jest łatwy do odczytywania przez ludzi. Takie pliki zawierają grafikę, kod, tekst, itd.

3.3.2 Określenie repozytorium dla przedmiotu (jako przykład)

1. Kod tworzony i uruchamiany w przykładach podczas zajęć będzie zapisywany w repozytorium na lokalnym dysku twardym, a w poniższym przykładzie konkretnie na pulpicie użytkownika.
2. Repozytorium można porównać do szafki w której zostaje umieszczony kod.
3. **Jupyter** otwiera szufladę, wyjmuje folder i prezentuje kod.
4. Kod można modyfikować, uruchamiać pojedyncze przykłady w folderze, dodawać nowe przykłady i w naturalny sposób komunikować się z kodem.

3.3.3 Definiowanie folderu roboczego dla przedmiotu (jako przykład)

1. Otwórz **Jupyter Notebook** z menu **Start**.
2. W panelu głównym znajdź i kliknij pozycję **Desktop**.
3. Z menu po prawej stronie wybierz **New/Folder**. **Jupyter** utworzy folder o nazwie **Untitled Folder**.
4. Zaznacz pole obok pozycji **Untitled Folder**.
5. Kliknij na górze strony przycisk **Rename**. Wyświetli się okno dialogowe **Reneme Directory**. Wpisz nazwę folderu (np. skrót nazwy przedmiotu).
6. Gotowe, masz repozytorium do gromadzenia plików i innych folderów, które będzie w "zasięgu" **Jupytera**.

3.4 Spyder

- Budowa środowiska
- Podstawowa obsługa środowiska,
 1. zmiana folderu roboczego
 2. tworzenie nowego pliku (skryptu) i jego zapis do wybranego folderu
 3. kodowanie wg utf-8
 4. znak # - komentarz w jednej linii

3.5 Google Colab

- <https://colab.research.google.com/notebooks/welcome.ipynb>
- Podstawowa obsługa środowiska,
 - przygotuj nowe środowisko w Navigatorze z właściwą wersją Pythona
 - wpisz w przeglądarce adres **drive.google.com** -> Otwórz Dysk -> Podaj login i hasło
 - utwórz nowy katalog w którym będą przechowywane pliki i otwórz ten katalog
 - wybierz: New -> More -> Google Colaboratory
 - przegląd środowisk uruchomieniowych:
 - pod przyciskiem Połącz wybrać opcję: *Połącz z hostowanym środowiskiem wykonawczym*
 - poczekać na połączenie
 - poleceniem: `!python --version` sprawdzić przydzieloną wersję Pythona
 - ponieważ zwykle przydzielane są starsze wersje, można połączyć się ze środowiskiem zainstalowanym na własnym hoście
 - w tym celu wybierz opcję: *Połącz się z lokalnym środowiskiem wykonawczym*
 - następnie w oknie dialogowym kliknij na podany link i wykonaj wymagane instalacje na lokalnym komputerze, wystartuj serwer i dokonaj autentykacji wykorzystując podane informacje przez serwer

3.6 Krótkie ćwiczenia wprowadzające do prezentowanych narzędzi

Ćwiczenie 1. Uruchomienie Pythona przy użyciu *Anaconda Prompt*.

1. Z menu **Start** otwórz okno **Anaconda Prompt**.
2. Uruchom Pythona: w oknie **Anaconda Prompt** napisz **Python** i przyciśnij **Enter**.
3. Wyświetli się znak zachęty `>>>`, co oznacza, że zgłosił się do pracy interpreter Pythona.
4. Za znakiem zachęty napisz:

```
print("Hello Anaconda")
```

i przyciśnij Enter.

5. Interpreter wykona polecenie i wyświetli na ekranie tekst **Hello Anaconda**

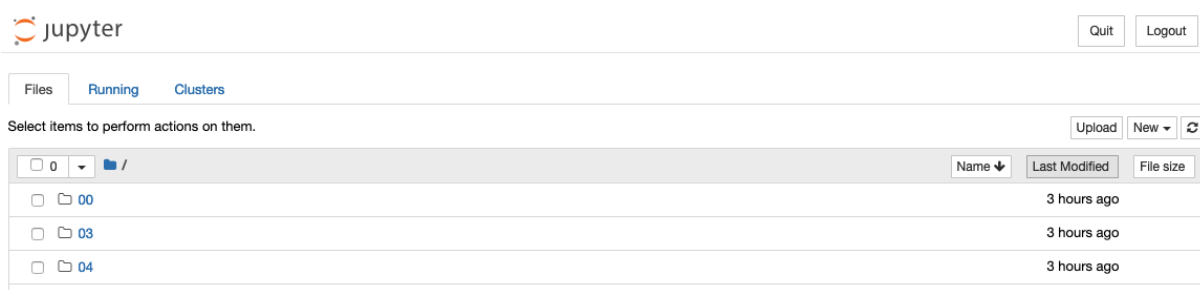
6. Wyjdź z Pythona: naciśnij kombinację **CTRL-Z**.

Ćwiczenie 2. Uruchomienie Pythona w programie *Jupyter Notebook*.

1. Na stronie głównej **Navigatora** w okienku **Aplikacje** po prawej stronie przewiń do kafelka **Jupyter Notebook**, zainstaluj go jeśli nie jest zainstalowany, a następnie uruchom.

2. Otworzy się nowa karta w domyślnej przeglądarce z **pulpitem nawigacyjnym notatnika (Notebook Dashboard)**.

Przykładowy obraz:



3. Po prawej stronie na górze znajduje się rozwijalne menu oznaczone opcją **New**. Utwórz nowy **Notebook** z wersją zainstalowanego **Pythona 3**.

4. W nowym **Notebooku** klikając bieżącą domyślną nazwę **Untitled** zmień nazwę pliku na:

tu wpisz kod przedmiotu; Przykład 1

np. **WdUM; Przykład 1** albo **CPO; Plik Testowy** albo **SI; Próba 1**

5. W pierwszym wierszu **Notebooka** wpisz polecenie:

```
print("Hello Anaconda")
```

6. Zapisz swój **Notebook** klikając **Save and Checkpoint** na pasku narzędziowym.

7. Uruchom program klikając przycisk **Run** lub wybierając z górnego menu **Cell -> Run All**

Przykładowy obraz:



Wynik wyświetla się w tej samej komórce co kod (kod znajduje się w prostokątnym polu, a jego wynik poza tym polem, ale zarówno kod jak i wynik wyświetlają się w tej samej komórce). W

środowisku **Jupyter Notebook** dane wyjściowe są wizualnie oddzielone od kodu, dzięki czemu można je odróżnić. Po uruchomieniu kodu **Jupyter Notebook** automatycznie utworzy nową komórkę.

8. Zamknij **Jupyter Notebook**: z górnego menu wybierz **File -> Close and Halt**.

Sterowanie powróci do strony głównej, gdzie widać nowy utworzony **Notebook** dodany do listy.

9. Kliknij przycisk **Quit** w prawej górnej części **Notebook Dashboard** i zamknij okno przeglądarki lub kartę z programem.

10. Zamknij program **Navigator**.

11. Inne możliwe do wykonywania czynności w **Notebooku**: Usuwanie notatnika, Eksportowanie notatnika, Importowanie notatnika.

Ćwiczenie 3. Program w Pythonie w aplikacji Spyder.

1. Otwórz **Navigator** -> z menu **Start** wybierz **Anaconda Navigator**

2. Ekran główny **Navigatora** wyświetla w kafelkach kilka aplikacji. Sprawdź czy program **Spyder** jest zainstalowany, jeśli nie jest to zainstaluj go.

3. **Spyder** IDE jest zintegrowanym środowiskiem programistycznym. Uruchom go.

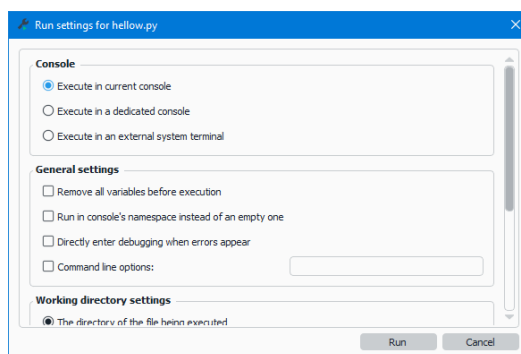
4. W programie **Spyder** w lewym panelu (nowy plik) usuń tekst zastępczy i wpisz polecenie:

```
print("Hello Anaconda")
```

5. W górnym menu kliknij **File – Save As**, nazwij program **hello.py** i zapisz go w dowolnie wybranej lokalizacji, np. w utworzonym wcześniej repozytorium dla przedmiotu.

6. Uruchom program klikając trójkątny przycisk **Run file**.

7. Jeśli pojawi się podobne okno jak poniżej, pozostaw opcje domyślne i kliknij przycisk **Run**.



8. Dane wyjściowe z Twojego programu wyświetlone zostaną w konsoli **Spydera** w prawym dolnym panelu.

9. Zamknij program **Spyder** -> Z menu **File** wybierz opcję **Quit**.

Ćwiczenie 4. Opcje uruchomieniowe programów Spyder i Jupyter Notebook z linii komend.

1. W oknie **Anaconda Prompt** napisz **spyder** i przyciśnij **Enter**. Uruchomi się program **Spyder** podobnie jak w przypadku gdy był wykorzystany program **Anaconda Navigator**.
2. Zamknij program **Spyder**.
3. W oknie **Anaconda Prompt** napisz **jupyter notebook** i przyciśnij **Enter**. Uruchomi się program **Jupyter Notebook** podobnie jak to miało miejsce wykorzystując program **Anaconda Navigator**.
4. Zamknij **Jupyter Notebook**.

4 Przegląd komponentów języka Python

4.1 Podstawy języka

4.1.1 Typy danych języka Python

Informacje wprowadzające:

- w Pythonie wszystko jest obiektem
- typy: łańcuchy znaków, liczby całkowite, liczby zmiennoprzecinkowe, liczby zespolone, typ logiczny
- obiekty tej samej klasy mają swoje własne stany i te same zachowania
- przy pomocy funkcji `dir` można podglądać wszystkie metody dostępne dla obiektów danej klasy,
przykład dla klasy `string`:
`zm_str = 'Python'`

przykład dla klasy `integer`:
`zm_int = 100`

przykład dla klasy `float`:
`zm_float = 10.5`

przykład dla klasy `bool`:
`zm_bool = True`
`print(dir(zm_bool))`

- Spyder pozwala podglądać typ zmiennej, a funkcja `type` pozwala programowo na sprawdzenie typu zmiennej:

```
type(zm_str)
str
type(zm_int)
int
type(zm_float)
float
type(zm_bool)
bool
type(True)
bool
```

1. Działania na liczbach i operacje logiczne

a. Typy wartości liczbowych i logicznych obsługiwane przez Pythona

Wszystkie liczby całkowite to dane typu **integer**, np. liczba 1 jest liczbą całkowitą, natomiast liczba 1.0 nie jest liczbą całkowitą bo ma część dziesiętną. W Pythonie liczby całkowite są reprezentowane przez typ **int**. Na większości platform dane typu **int** pozwalają przechowywać liczby w zakresie:

od -9 223 372 036 854 775 808 **do** 9 223 372 026 854 775 807

których wartości mieszczą się w zmiennej 64-bitowej.

Wszystkie liczby zawierające część dziesiętną to wartości **zmiennoprzecinkowe**. Na przykład 1.0 ma część dziesiętną, więc jest to liczba **zmiennoprzecinkowa**. Python przechowuje wartości **zmiennoprzecinkowe** w typie danych **float**. Na większości platform maksymalna wartość, jaką można zapisać w zmiennej **zmiennoprzecinkowej** wynosi: $\pm 1,7976931348623157 \times 10^{308}$ a minimalna wartość to $\pm 2,2250738585072014 \times 10^{-308}$

Liczba zespolona składa się z części rzeczywistej i części urojonej połączonych w parę. Część urojona liczby zespolonej jest oznaczana literą **j**. Aby utworzyć liczbę zespoloną, w której część rzeczywista to 3, a część urojona to 4 można wykonać następujące podstawienie:
`myComplex = 3 + 4j.`

Argumenty logiczne wymagają wartości boolowskich, których nazwa pochodzi od nazwiska Georg'a Boola. Kiedy używamy wartości logicznych w Pythonie posługujemy się typem **bool**. Zmienna tego typu może zawierać tylko dwie wartości: **True** lub **False**. Wartość do zmiennej typu **bool** można przypisać za pomocą słów kluczowych **True** lub **False** albo można utworzyć wyrażenie, które definiuje logiczny warunek o wartości równej **True** (prawda) lub **False** (fałsz). Na przykład można użyć wyrażenia `myBool = 1 > 2`, co odpowiada wartości **False**, ponieważ 1 nie jest większe niż 2.

Po tym wstępie na temat danych liczbowych i logicznych w poniższych punktach nastąpi szybka prezentacja sposobu pracy z danymi numerycznymi i logicznymi.

b. Przypisywanie wartości do zmiennych liczbowych

Podczas pisania kodu programu dane przechowuje się w zmiennych. Tak więc chcąc przetwarzać dane należy używać zmiennych, umieszczać w nich nowe dane, modyfikować je jeśli jest taka potrzeba. Modyfikacja informacji polega na tym, że najpierw trzeba uzyskać dostęp do zmiennej, a następnie zapisać w niej nową wartość.

Aby zapisać dane w zmiennej należy przypisać do niej dane za pomocą dowolnego operatora przypisania – czyli specjalnego symbolu informującego o sposobie przechowywania danych.

Poniżej znajduje się wykaz wszystkich operatorów przypisania używanych w Pythonie.

Tabela 1. Operatory przypisania w Pythonie

Operator	Opis	Przykład
=	Przypisuje wartość prawego operandu do lewego operandu	<code>MyVar = 5</code> Zmienna <code>MyVar</code> otrzymuje wartość 5
+=	Dodaje wartość zapisaną w prawym operandzie do wartości w lewym operandzie i umieszcza wynik w lewym operandzie	<code>MyVar += 2</code> Zmienna <code>MyVar</code> otrzymuje wartość 7
-=	Odejmuje wartość zapisaną w prawym operandzie od wartości zapisanej w lewym operandzie i umieszcza wynik w lewym operandzie	<code>MyVar -= 2</code> Zmienna <code>MyVar</code> otrzymuje wartość 3
*=	Mnoży wartość zapisaną w prawym operandzie przez wartość zapisaną w lewym operandzie i umieszcza wynik w lewym operandzie	<code>MyVar *= 2</code> Zmienna <code>MyVar</code> otrzymuje wartość 10
/=	Dzieli wartość zapisaną w prawym operandzie do wartości zapisaną w lewym operandzie i umieszcza wynik w lewym operandzie	<code>MyVar /= 2</code> Zmienna <code>MyVar</code> otrzymuje wartość 2.5
%=	Dzieli wartość zapisaną w prawym operandzie do wartości zapisaną w lewym operandzie i umieszcza resztę w lewym operandzie	<code>MyVar %= 2</code> Zmienna <code>MyVar</code> otrzymuje wartość 1
**=	Oblicza wartość lewego operandu podniesionego do potęgi równej wartości w prawym operandzie i umieszcza wynik w lewym operandzie	<code>MyVar **= 2</code> Zmienna <code>MyVar</code> otrzymuje wartość 25
//=	Dzieli wartość zapisaną w lewym operandzie przez wartość zapisaną w prawym operandzie i umieszcza całkowitoliczbowy wynik w lewym operandzie	<code>MyVar //= 2</code> Zmienna <code>MyVar</code> otrzymuje wartość 2

c. Wykonywanie działań arytmetycznych

Przechowywanie danych w zmiennych jest bardzo użyteczne, ponieważ są one łatwo dostępne i można za ich pomocą wykonywać różne działania arytmetyczne. Python obsługuje typowe operatory arytmetyczne, których używa się do ręcznego wykonywania zadań.

Tabela 2. Operatory arytmetyczne w Pythonie

Operator	Opis	Przykład
+	Dodaje do siebie dwie wartości	$5 + 2 = 7$
-	Odejmuje prawy operand od lewego	$5 - 2 = 3$
*	Mnoży prawy operand przez lewy	$5 * 2 = 10$
/	Dzieli lewy operand przez prawy	$5 / 2 = 2.5$
%	Dzieli lewy operand przez prawy operand i zwraca resztę	$5 \% 2 = 1$
**	Oblicza wartość potęgi; lewy operand to podstawa, prawy to wykładnik	$5 ** 2 = 25$
//	Wykonuje dzielenie całkowitoliczbowe, w którym lewy operand jest dzielony przez prawy operand, a w wyniku zwracana jest tylko liczba całkowita	$5 // 2 = 2$

Czasami działania dotyczą tylko jednej zmiennej. Python obsługuje operatory **jednoargumentowe**, tzn. takie, które wykonują działania tylko na jednej zmiennej.

Tabela 3. Operatory jednoargumentowe w Pythonie

Operator	Opis	Przykład
~	Odwraca bity w liczbie w taki sposób, że wszystkie bity 0 przyjmują wartość 1 i odwrotnie	$\sim 4 = -5$
-	Neguje wartość argumentu w taki sposób, że liczba dodatnia staje się ujemna i odwrotnie	$-(-4)$ zwraca 4 -4 zwraca -4
+	Ten operator istnieje wyłącznie w celu zapewnienia kompletności; zwraca tę samą wartość, którą podamy jako argument wejściowy	wyrażenie +4 zwraca 4

Komputery mogą wykonywać inne rodzaje zadań matematycznych ze względu na sposób działania procesora. Należy pamiętać, że w komputerach dane są przechowywane w postaci ciągu

pojedynczych bitów. Python umożliwia dostęp do pojedynczych bitów za pomocą operatorów bitowych.

Tabela 4. Operatory bitowe w Pythonie

Operator	Opis	Przykład
& Koniunkcja bitowa	Określa, czy oba bity w obrębie dwóch operandów mają wartość 1, i kiedy tak jest, ustawia bit wynikowy na 1	0b1100 & 0b0110 = 0b0100
 Alternatywa bitowa	Określa, czy którykolwiek bit z dwóch operandów ma wartość 1, i kiedy tak jest, ustawia bit wynikowy na 1	0b1100 0b0110 = 0b1110
^ Bitowa różnica symetryczna	Określa, czy dokładnie jeden bit w obrębie dwóch operandów ma wartość 1, natomiast gdy oba mają wartość 1 lub gdy oba bity mają wartość 0, wynik ma wartość 0	0b1100 ^ 0b0110 = 0b1010
~ Uzupełnienie do jedności	Oblicza wartość uzupełnienia liczby do jedności	~0b1100 = -0b1101 ~0b0110 = -0b0111
<< Przesunięcie w lewo	Przesuwa bity w lewym operandzie w lewo o wartość prawego operandu wszystkie nowe bity są ustawiane na 0, a wszystkie bity, które wychodzą poza zakres są tracone	0b00110011 << 2 = 0b11001100
>> Przesunięcie w prawo	Przesuwa bity w lewym operandzie w prawo o wartość prawego operandu; wszystkie nowe bity są ustawione na 0, a wszystkie bity, które wychodzą poza zakres są tracone	0b00110011 >> 2 = 0b00001100

d. Porównywanie danych za pomocą wyrażeń boolowskich

Używanie arytmetyki do modyfikowania zawartości zmiennych jest rodzajem działań na danych.

Aby określić efekt działania na danych, komputer musi porównać bieżący stan zmiennej ze stanem pierwotnym lub stanem znanej wartości. W niektórych przypadkach konieczne jest również wykrycie statusu jednego wejścia względem innego. Wszystkie te operacje sprawdzają relacje pomiędzy dwiema zmiennymi, więc operatory służące do ich wykonywania są operatorami relacyjnymi.

Tabela 5. Operatory relacyjne w Pythonie

Operator	Opis	Przykład
<code>==</code>	Określa czy dwie wartości są sobie równe	Wyrażenie <code>1 == 2</code> ma wartość <code>False</code>
<code>!=</code>	Określa czy dwie wartości nie są sobie równe	Wyrażenie <code>1 != 2</code> ma wartość <code>True</code>
<code>></code>	Sprawdza czy wartość lewego operandu jest większa niż wartość prawego operandu	Wyrażenie <code>1 > 2</code> ma wartość <code>False</code>
<code><</code>	Sprawdza czy wartość lewego operandu jest mniejsza niż wartość prawego operandu	Wyrażenie <code>1 < 2</code> ma wartość <code>True</code>
<code>>=</code>	Sprawdza czy wartość lewego operandu jest większa bądź równa wartości prawego operandu	Wyrażenie <code>1 >= 2</code> ma wartość <code>False</code>
<code><=</code>	Sprawdza czy wartość lewego operandu jest mniejsza bądź równa wartości prawego operandu	Wyrażenie <code>1 <= 2</code> ma wartość <code>True</code>

Czasami użycie operatora relacyjnego nie pozwala uzyskać pełnego obrazu porównania dwóch wartości. Na przykład może być konieczne sprawdzenie warunku, w którym potrzebne są dwa oddzielne porównania, takie jak `MyAge > 40` i `MyHeight < 74`. Potrzeba dodania warunków do porównania wymaga zastosowania jakiegoś operatora logicznego. Operatory logiczne zostały zestawione w kolejnej tabeli.

Zapewnienie określonego porządku porównań zapewnia się dzięki nadaniu niektórym operatorom większej wagi od innych. Kolejność stosowania operatorów określa się terminem pierwszeństwa operatorów. Pierwszeństwo operatorów wszystkich popularnych operatorów Pythona zestawiono w kolejnej tabeli. W tabeli uwzględniono kilka operatorów, których dotąd jeszcze nie omówiono. Przy porównaniach zawsze należy brać pod uwagę pierwszeństwo operatorów, ponieważ jeśli się tego nie zrobi, to założenia dotyczące wyniku porównania prawdopodobnie będą błędne.

Tabela 6. Operatory logiczne w Pythonie

Operator	Opis	Przykład
<code>and</code>	Określa czy oba operandy są prawdziwe	<code>True and True</code> ma wartość <code>True</code> <code>True and False</code> ma wartość <code>False</code> <code>False and True</code> ma wartość <code>False</code> <code>False and False</code> ma wartość <code>False</code>
<code>or</code>	Określa czy jeden z dwóch argumentów jest prawdziwy	<code>True or True</code> ma wartość <code>True</code> <code>True or False</code> ma wartość <code>True</code> <code>False or True</code> ma wartość <code>True</code>

		False or False ma wartość False
not	Neguje wartość prawdy pojedynczego argumentu; wartość True przekształca na False i odwrotnie	not True ma wartość False not False ma wartość True

Tabela 7. Pierwszeństwo stosowania operatorów w Pythonie

Operator	Opis
()	Nawiasów używamy w celu grupowania wyrażeń i nadpisywania ich domyślnego priorytetu, dzięki czemu możemy wymusić, aby działanie o niższym priorytecie (np. dodawanie) było wykonywane przed działaniem o wyższym priorytecie (np. mnożeniem)
**	Potęgowanie podnosi wartość lewego argumentu do potęgi określonej za pomocą prawego argumentu.
~ + -	Operatory jednoargumentowe wykonują działania na jednej zmiennej lub wyrażeniu
* / % //	Mnożenie, dzielenie, modulo, i dzielenie całkowitoliczbowe
+ -	Dodawanie i odejmowanie
>> <<	Przesunięcie bitowe w prawo i w lewo
&	Bitowy operator AND
^	Bitowa różnica symetryczna i standardowy operator OR
<= < > >=	Operatory porównania
== !=	Operatory równości
= %= /= //= -= += *= **=	Operatory przypisania
is is not	Operatory tożsamości
in not in	Operatory członkostwa
not or and	Operatory logiczne

2. Tworzenie ciągów znaków i posługiwanie się nimi

Ciągi znaków są łatwymi do zrozumienia dla ludzi typami danych, ale nie są zrozumiałe dla komputerów.

Ciąg znaków lub inaczej **łańcuch znaków** (ang. string) to dowolna grupa znaków, które umieszczane są w cudzysłowie. Na przykład `myString = "Python to fajny język."` przypisuje ciąg znaków do zmiennej `myString`.

Głównym powodem stosowania ciągów znaków podczas analizowania danych, pracy z obrazami lub wykorzystywania technik uczenia maszynowego jest zapewnienie interakcji z użytkownikiem, jako żądanie wprowadzania danych wejściowych albo jako mechanizm ułatwiający zrozumienie wyników. W rzeczywistości komputer w ogóle nie widzi liter, a każda litera używana w programie jest w pamięci komputera reprezentowana przez liczbę. Na przykład litera A jest liczbą 65, co można zobaczyć wpisując w środowisku interpretera Pythona polecenie `ord("A")`. Za pomocą polecenia `ord()` można przekształcić na liczbowy odpowiednik każdą literę.

Ponieważ komputer nie rozumie ciągów znaków, do ich konwersji na liczbę można używać funkcji **int()** lub **float()**. Na przykład polecenie `myInt = int("123")` tworzy zmienną typu `int` o nazwie `myInt` zawierającą wartość 123.

Liczbę można także przekształcać na ciąg znaków. Służy do tego polecenie **str()**. Na przykład jeśli wpisujemy `myStr = str(1234.56)` to utworzymy ciąg znaków zawierający wartość "1234.56" przypisany do zmiennej `myStr`.

A zatem można z łatwością konwertować ciągi znaków na liczby oraz liczby na ciągi znakowe.

Podobnie jak w przypadku liczb, w odniesieniu do ciągów znaków (a także wielu innych obiektów) można użyć specjalnych operatorów. **Operatory członkostwa** umożliwiają sprawdzenie, czy ciąg znaków zawiera określoną treść.

Tabela 8. Operatory członkostwa w Pythonie

Operator	Opis	Przykład
<code>in</code>	Określa czy wartość w lewym operandzie występuje w ciągu znajdującym się w prawym operandzie	Wyrażenie "Witaj" in "Witaj i żegnaj" zwraca True
<code>not in</code>	Określa czy wartości w lewym operandzie brakuje w ciągu znajdującym się w prawym operandzie	Wyrażenie "Witaj" not in "Witaj i żegnaj" zwraca False

Z powyższego opisu wynika, że trzeba znać rodzaj danych zapisanych w zmiennych. W tym celu można użyć operatorów tożsamości, które zostały zestawione w tabeli poniżej.

Tabela 9. Operatory tożsamości w Pythonie

Operator	Opis	Przykład
<code>is</code>	Przyjmuje wartość True gdy typ wartości lub wyrażenia w prawym operandzie wskazuje na ten sam typ, który ma lewy operand	Wyrażenie <code>type(2) is int</code> ma wartość True
<code>not is</code>	Przyjmuje wartość True gdy typ wartości lub wyrażenia w prawym operandzie wskazuje na ten inny typ niż typ, który ma lewy operand	Wyrażenie <code>type(2) is not int</code> ma wartość False

1. Dostęp do dokumentacji dotyczących metod dla typu `string`:

```
text = 'To jest podstawowy kurs Pythona.\nPython jest cool!'
print(text)
```

To jest podstawowy kurs Pythona.

Python jest cool!

```
help(str.count)
```

Help on method_descriptor:

```
count(...)
```

```
S.count(sub[, start[, end]]) -> int
```

Return the number of non-overlapping occurrences of substring sub in

string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

2. Przegląd wybranych metod stosowanych na zmiennych tekstowych:

- Zamiana pierwszej litery w ciągu znakowym na dużą:

```
language = 'python'
language.capitalize()
```

Python

- Zamiana pierwszej litery w każdym wyrazie w ciągu znakowego na dużą literę:

```
text.title()
```

```
'To Jest Podstawowy Kurs Pythona.\nPython Jest Cool!'
```

- Zliczanie liczby wystąpień podanego ciągu znaków:

```
text.count('Python')
```

```
2
```

- Kontrola początku lub końca ciągu znakowego:

```
language.startswith('Py')
```

```
True
```

```
text.endswith('!')
```

```
True
```

Wykorzystana często do sprawdzania rozszerzeń plików w folderze, lub do ich odfiltrowywania, np.:

```
'plik_tekstowy.txt'.endswith('.txt')
```

```
True
```

```
'plik_tekstowy.txt'.endswith('.py')
```

```
False
```

- Poszukiwanie w ciągu znakowym indeksu od którego zaczyna się dany podciąg:

```
text.find('Python')
```

```
24
```

Możliwe jest sprytnie wykorzystanie tej funkcji do wycinania fragmentu ciągu:

```
text[text.find('Python'):]
```

```
'Pythona.\nPython jest cool!'
```

- Sprawdzenie czy zmienna ma wyłącznie znaki alfanumeryczne:

```
'alfa123'.isalnum()
```

```
True
```

```
'alfa123 '.isalnum()
```

```
False
```

```
'alfa123_'.isalnum()
```

```
False
```

- Sprawdzenie czy wszystkie znaki w ciągu są cyframi:

```
'12345'.isdigit()
```

```
True
```



```
'12345f'.isdigit()
```

False

- Sprawdzenie czy wszystkie litery w ciągu są małymi literami:

```
'Python'.islower()
```

True

```
'Python'.islower()
```

False

```
'python3.9'.islower()
```

True

- Sprawdzenie czy wszystkie litery w ciągu są dużymi literami:

```
'PYTHON'.isupper()
```

True

```
'Python'.isupper()
```

False

```
'PYTHON3.9'.isupper()
```

True

- Łączenie elementów listy przy pomocy metody join:

```
' '.join(['Python', '3.9'])
```

'Python 3.9'

```
'.'.join(['www', 'amw', 'gdynia', 'pl'])
```

'www.amw.gdynia.pl'

```
', '.join(['10', '20', '30', '40', '50'])
```

- Zamiana określonego znaku na inny w zadanym ciągu znakowym:

```
'10#20#30#40#50'.replace('#', ',')
```

'10,20,30,40,50'

- Wycinanie białych znaków z tekstu:

```
' Python '.strip()
```

'Python'

```
' Python '.lstrip()
```

'Python '

```
' Python '.rstrip()
```

' Python'

- Rozdzielenie łańcucha znakowego na elementy listy wykorzystując określony znak, domyślnym znakiem jest spacja:

```
'1,2,3,4,5,6,7,8,9,10'.split(',')  
['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']  
  
'c++ java php Python'.split()  
['c++', 'java', 'php', 'Python']
```

- Wypełnienie tekstu zerami nieznaczącymi:

```
'123'.zfill(7)  
'0000123'
```

3. Działania na datach

Podobnie jak w przypadku ciągów znaków, komputery nie rozumieją dat, a data i godzina z punktu widzenia komputera nie istnieje. To program ma interpretować dane aby zorganizować właściwe przetwarzanie z wykorzystaniem dat i godzin.

Uwaga!

Aby móc pracować z datami i godzinami należy zaimportować odpowiedni moduł wydając polecenie **import datetime**.

Aby uzyskać aktualny czas należy wpisać polecenie `datetime.datetime.now()`. Wyświetlą się pełne informacje na temat daty i godziny, na przykład:

```
datetime.datetime(2021, 9, 16, 13, 21, 31, 271238)
```

Wyniki można formatować, np. `str(datetime.datetime.now().date())` wyświetli '2021-09-16'

W Pythonie jest także polecenie `time()`, które można użyć aby uzyskać aktualną godzinę. Można otrzymać oddzielne wartości dla każdego ze składników, które tworzą datę i godzinę, używając wartości: `day`, `month`, `year`, `hour`, `minute`, `second` i `microsecond`.

4.1.2 Zmienne i ich konwencje w Pythonie

1. nazwy zmiennych zaczynają się od liter i mogą zawierać cyfry (są to typowe zmienne):

```
imie = 'Michał'
```

zmienna `imie` jest widoczna w przestrzeni nazw Spydera

2. zmienne ukryte:

```
_imie = 'Antek'
```

zmienna `_imie` istnieje (w konsoli można ją wyświetlić), ale nie jest widoczna w przestrzeni nazw Spydera

3. zalecany sposób w Pythonie używania nazw dla:

- a. **zmiennych i funkcji** – to unikanie pojedynczych liter w miejsce pełnych nazw używanych wyrazów z łączeniem ich znakiem podkreślenia – czyli preferowany jest styl **snake_case**: `x`, `my_variable`, `function`, `my_function`
- b. **stałych** – to stosowanie dużych liter – czyli preferowany styl **UPPER**: `CONSTANT`, `MY_CONSTANT`
- c. **klas** – styl **pascal_case**: `Model`, `MyClass`
- d. nie używać słów kluczowych Pythona, które można pobrać przez polecenia:

```
import keyword  
keyword.kwlist  
['False',  
 'None', ...
```

4.1.3 Instrukcja `print()`

1. konsola służy do interaktywnych działań – są to dane ulotne, czyli po wyłączeniu środowiska dane gubimy
przykład: `5 + 5` – wyświetli wynik w konsoli
2. utworzenie pliku `01_wprowadzenie.py`; instrukcje zawarte w pliku można zapisać i później odtworzyć
przykład: `5 + 5` – uruchomienie pliku nie wyświetli wyniku w konsoli poza informacją, że został uruchomiony plik w danej lokalizacji
3. aby otrzymać wynik w konsoli trzeba zastosować instrukcję `print()`:
`print(5 + 5)` – wyświetli wynik w konsoli
4. instrukcja `print()` działa także w konsoli: `print(5 + 5)` – wyświetli wynik w konsoli
5. podzielenie skryptu na mniejsze części, tworzenie komórek w skrypcie – kolejno znaki: `# %%` co powoduje, że można traktować powstałe komórki jako oddzielne skrypty, uruchamiane oddzielnie za pomocą przycisku z podwójną zieloną strzałką na pasku narzędziowym lub przez skrót `Shift + Enter`
6. kolejna komórka powstaje poprzez kolejne wprowadzenie znaków `# %%` w nowej linii
7. `print(10 * 10)` - zwrócić uwagę na spacje przed i po operatorze – jest to umowna konwencja w Pythonie poprawiająca czytelność kodu.
8. uruchomienie komórki zawierającej dwie instrukcje
`print(5 + 5)`

```
print(10 * 10)
```

w konsoli wyświetli dwa wyniki: 10 i 100 w oddzielnych wierszach

9. ale jeżeli wpisujemy w jednej komórce polecenia

```
5 + 5 * 10
```

```
50 - 5 * 10
```

to w konsoli wyświetli się wynik tylko ostatniej instrukcji: **0**

10. jeśli w komórce będzie taka kombinacja:

```
print(5 + 5 * 10)
```

```
50 - 5 * 10
```

wyświetlone zostaną oba wyniki, ale pierwszy jako wynik w komórce konsoli `In` a drugi w komórce konsoli `Out`

11. zwykle będziemy pomijać instrukcję `print()` działając na w trybie interaktywnym skryptu (wykorzystując komórki)

12. kolejność działań zgodna z zasadami matematyki:

```
(100 - 5 ** 2) * 2
```

 – wynik: **150**

- Tworzenie zmiennych numerycznych i proste obliczenia

13. trzeba wiedzieć, że przy każdym uruchomieniu Spydera tworzone jest tzw. jądro z którym współpracuje Python, które utrzymuje wartości wszystkich zmiennych

14. podczas pracy ze środowiskiem jądro można restartować czyszcząc jednocześnie pamięć zmiennych

15. czyszczenie konsoli – polecenie `clear` (nie usuwa zmiennych z pamięci roboczej)

16. przykład instrukcji przypisania:

```
x = 10
```

jej uruchomienie w komórce skryptu powoduje wydruk instrukcji w komórce `In` konsoli i pojawienie się zmiennej `x` w obszarze roboczym Spydera (opcja: Variable Explorer), gdzie widać jej nazwę, typ, rozmiar i wartość

17. w tej samej komórce dodajemy instrukcję przypisania dla drugiej zmiennej:

```
x = 10
```

```
y = 20
```

(zwrócić uwagę na spacje)

18. polecenie `x * y` zostaje wykonane i wynik pojawia się w konsoli

19. polecenie `z = x * y` oblicza wartość mnożenia zmiennych `x` i `y`, a wynik podstawia do zmiennej `z` (wyniku nie wyświetla)

20. wyświetlenie wyniku nastąpi po użyciu instrukcji `print(z)`

- tworzenie zmiennych tekstowych i ich obsługa

21. używanie apostrofów lub cudzysłowów

```
'Wiwat Python'
```

```
"Wiwat Python"
```

22. `"Wiwat kod Python'a"`

lub ze znakiem ucieczki `\`:

```
'Wiwat kod Python\'a'
```

23. z użyciem instrukcji `print()`

```
print('Wiwat Python')
```

24. z użyciem znaku nowej linii `\n`

```
print('Wiwat\nPython')
```

25. wydruk z tabulacją

```
print('\tWiwat Python')
```

26. uwaga na stosowanie `\n` i `\t`

```
print('C:\path\to\somewhere\next')
```

```
C:\path      o\somewhere
ext
```

poprawnie ze znakiem ucieczki:

```
print('C:\\path\\to\\somewhere\\next')
```

poprawnie z parametrem `r`, czyli `raw text` (surowy tekst):

```
print(r'C:\path\to\somewhere\next')
```

poprawnie tak jak robi to Python, opcja rekomendowana:

```
print('C:\\path\\to\\somewhere\\next')
```

```
C:\path\to\somewhere\next
```

27. pobranie adresu ścieżki aktualnego katalogu roboczego:

```
import os
```

```
os.getcwd()
```

```
'C:\\PYTHON\\Katalog roboczy\\wprowadzenie'
```

28. wydruk z użyciem trzech znaków `"""`

```
print("""Wiwat
```

```
Python""")
```

```
Wiwat
```

```
Python
```

29. możliwość stworzenia instrukcji do uruchomienia programu

```
print("""Instrukcja uruchomienia pliku example.py
    -- file [file name]
        zapisuje output do pliku

    -- quiet
        wycisza logi w konsoli
Koniec""")
```

Instrukcja uruchomienia pliku example.py

```
-- file [file name]
    zapisuje output do pliku

-- quiet
    wycisza logi w konsoli
```

Koniec

30. drukowanie wartości zmiennej, drukowanie wielokrotne

```
text_var = 'Wiwat Python. '
print(text_var * 3)
print('*' * 30)
```

Wiwat Python. Wiwat Python. Wiwat Python.

31. domyślne łączenie łańcuchów w Pythonie

```
print('Py' 'thon')
```

Python

32. wiersze nie powinny być dłuższe niż 79 kolumn, przykładowo

```
url = 'https://helion.pl/ksiazki/Python-instrukcje-dla-programisty-
wydanie-ii-eric-matthes,blkpy2.htm#format/e'
url_2 = (' https://helion.pl/ksiazki/'
'Python-instrukcje-dla-programisty-wydanie-ii-eric-matthes,'
'blkpy2.htm#format/e')
```

sprawdzić w zmiennych co zawierają zmienne url i url_2

33. łączenie łańcuchów:

```
name = 'Python'
print(name + '3.9')
```

Python3.9

lub z dodaną spacją:

```
name = 'Python'
print(name + ' 3.9')
```

Python 3.9

lub przez sklejenie w instrukcji print():

```
name = 'Python'
print(name, '3.9')
```

Python 3.9

34. łączenie łańcuchów i liczb:

```
nazwisko = 'Żółtko'
wiek = 23
print(nazwisko + wiek)
```

TypeError: can only concatenate str (not "int") to str

po konwersji liczby na łańcuch za pomocą funkcji str():

```
print(nazwisko + str(wiek))
```

Żółtko23

lub bardziej elegancko:

```
print(nazwisko, wiek)
```

Żółtko 23

i najbardziej elegancko stosując metodę formatowania:

```
tytul = 'Pan'
imie = 'Michał'
nazwisko = 'Żółtko'
wiek = 23
print('{0} {1} {2} ma {3} lata'.format(tytul, imie, nazwisko, wiek))
```

Pan Michał Żółtko ma 23 lata

ze zmianą kolejności indeksów zmiennych:

```
print('{0} {2} {1} ma {3} lata'.format(tytul, imie, nazwisko, wiek))
```

35. inne przykłady prostych działań i obliczeń

```
pixel = 100
pixel += 50
pixel /= 255
print(pixel)
```

0.5882352941176471

```
imie = 'Michał'
nazwisko = 'Żółtko'
```

```
osoba = imie + nazwisko
```

w pamięci zmienna osoba ma wartość: **MichałŻółtko**

4.1.4 Instrukcja input()

1. wprowadzanie danych, interakcje:

```
imie = input('Podaj imię: ')\nprint('Witaj', imie)
```

Podaj imię: Michał

Witaj Michał

2. kolejny przykład, formatowanie wyjścia przy użyciu metody format:

```
imie = input('Podaj imię: ')\nprint('Witaj {}'.format(imie))
```

Podaj imię: Michał

Witaj Michał!

3. kolejny przykład:

```
imie = input('Podaj imię: ')\njezyk = input('Podaj Twój ulubiony język programowania: ')\nprint('Cześć {}! Lubisz pisać w języku {}'.format(imie, jezyk))
```

Podaj imię: Michał

Podaj Twój ulubiony język programowania: Java

Cześć Michał! Lubisz pisać w języku Java

4. liczby są wprowadzane jako łańcuchy, potrzebna jawna konwersja

```
imie = input('Podaj imię: ')\nwiek = input('Podaj swój wiek: ')\nprint('Cześć {}! Masz {} lat'.format(imie, wiek))
```

Podaj imię: Michał

Podaj swój wiek: 23

Cześć Michał! Masz 23 lat

Zmienna wiek jest typu str

Konwersja str na int:

```
wiek = int(input('Podaj swój wiek: '))
```

4.1.5 Operator wycinania [start:stop:step]

3. pozwala pozyskiwać fragmenty danych ze zmiennych łańcuchowych, list, krotek i innych struktur danych

4. przy pomocy indeksu ze zmiennej łańcuchowej `name` wycinamy fragmenty (indeksy liczone są od 0):

```
name = 'Python'
print(name)
print(name[0])      # wydruk pierwszej litery
print(name[5])      # wydruk ostatniej litery
print(name[-1])     # wydruk ostatniej litery
print(name[-2])     # wydruk przedostatniej litery
print(name[1:4])    # UWAGA!!!: wycięcie od indeksu 1 do 3
print(name[:4])     # wycięcie od początku do indeksu 3
print(name[2:])     # wycięcie od indeksu 2 do końca
print(name[:])      # pobiera cały łańcuch
print(name[-3:])    # pobiera trzy ostatnie elementy
```

Python

P

n

n

o

yth

Pyth

thon

Python

hon

5. inny przykład:

```
zmienna = 'Python to fajny język'
print(zmienna[::2])
```

Pto ofjyjzk

6. kolejny przykład, z liczbami:

```
liczby = '1,2,3,4,5,6,7,8,9,0'
print(liczby[::2])
```

1234567890

7. kolejny przykład, krok liczony od końca:

```
name = 'Python'
print(name[::-1])
```

nohtyP

8. wzmianka o operatorze in:

```
name = 'Python'
```

```
'P' in name
```

```
True
```

```
'p' in name
```

```
False
```

```
'Py' in name
```

```
True
```

4.2 Moduły wbudowane

1. **datetime**

- a. dostarcza narzędzi do pracy z datami i czasem
- b. `help(datetime)`

2. **os**

- a. dostarcza łatwą interakcję z systemem operacyjnym
- b. zawiera wiele przydatnych funkcji do pracy z systemem plików

3. **sys**

4. **copy**

5. **string**

6. **re**

7. **collections**

8. **pprint**

9. **pathlib**

10. **random**

11. **math**

12. **statistics**

13. **numbers**

14. **decimal**

15. **fractions**

16. **itertools**

17. **functools**

18. **pickle**

19. **zipfile**

20. **csv**

21. **json**

22. **xml**

23. **urllib.request**

24. **timeit**

25. **keyword**

26. **operator**

4.3 Struktury danych: listy, krotki, słowniki, zbiory

4.3.1 Listy

- uporządkowana struktura elementów
- kolekcja elementów ułożonych w określonej kolejności
- po utworzeniu można ją modyfikować
- przechowuje elementy różnych typów danych
- jest najpopularniejszą strukturą danych w Pythonie (obok słowników)
- nawias kwadratowy [] wskazuje listę, a jej elementy rozdzielone są przecinkami

1. tworzenie pustej listy:

```
pusta_lista = list()
print(pusta_lista)

[]

druga_pusta_lista = []
```

2. tworzenie listy bezpośrednio:

```
systems = ['Linux', 'Windows', 'Apple iOS', 'Android', 'Chrome OS']
(sprawdzić w Variable explorer)
print(type(systems))

<class 'list'>
```

3. zmiana wartości elementu listy:

```
systems[1] = 'Windows 10'
print(systems)

['Linux', 'Windows 10', 'Apple iOS', 'Android', 'Chrome OS']
```

4. tworzenie różnych list:

```
lista_liczb = [5, 10, 15, 101]
print(lista_liczb)

[5, 10, 15, 101]

lista_mieszana = ['Python', 3.9, 'Windows', 10, True, False]
print(lista_mieszana)

['Python', 3.9, 'Windows', 10, True, False]
```

5. zagnieżdżanie list:

```
listy_w_liscie = [[2019, [2020, 'XT'], 30], ['Linux', 'Windows', 'QNX'], 2021]
print(listy_w_liscie)

[[2019, [2020, 'XT'], 30], ['Linux', 'Windows', 'QNX'], 2021]
```

6. łączenie list – lista zagnieżdżona:

```
hardware = ['x86', 'ARP', 'PowerPC', 'SPARC']
software = ['Windows', 'Linux', 'FreeBSD', 'Solaris']
platforms = [hardware, software]
print(platforms)
[['x86', 'ARP', 'PowerPC', 'SPARC'], ['Windows', 'Linux', 'FreeBSD', 'Solaris']]
print(len(platforms))
2
```

7. inny sposób łączenia list – w jedną listę:

```
pl = hardware + software
print(pl)
['x86', 'ARP', 'PowerPC', 'SPARC', 'Windows', 'Linux', 'FreeBSD', 'Solaris']
print(len(pl))
8

pl += ['QNX']
print(pl)
['x86', 'ARP', 'PowerPC', 'SPARC', 'Windows', 'Linux', 'FreeBSD', 'Solaris', 'QNX']
```

8. wycinanie list:

- przykłady list:

```
lista = [11, 22, 33, 44, 55, 66]
indx_w_przod = [0, 1, 2, 3, 4, 5,]
indx_od tylu = [-6, -5, -4, -3, -2, -1]
```

- możliwe schematy wycinania:

lista[start:stop]

lista[indx]

lista[start:]

lista[:stop]

lista[::krok]

- przykłady:

```
lista[0]      -> 11
lista[1]      -> 22
lista[-1]     -> 66
lista[-5]     -> 22
lista[:4]     -> [11, 22, 33, 44]
lista[1:5]    -> [22, 33, 44, 55]
```

```
lista[-5:-1] -> [22, 33, 44, 55]
lista[::-1] -> [66, 55, 44, 33, 22, 11]
```

9. metody stosowane podczas używania list:

- utworzenie pustej listy:

```
techno = []
print(techno) -> []
```

- `append()` - dodawanie elementu do listy na jej koniec:

```
techno.append('Python')
techno.append('java')
techno.append('java')
print(techno)
['Python', 'java', 'java']
techno.append(['QNX', 'BSD'])
print(techno)
['Python', 'java', 'java', ['QNX', 'BSD']]
```

- `extend()` – dodawanie elementów do listy na tym samym poziomie:

```
techno = ['Python', 'java']
techno.extend(['SQL', 'C++'])
print(techno)
['Python', 'java', 'SQL', 'C++']
```

- `insert()` – dodawanie elementów do listy z określonym indeksem:

```
techno.insert(0, 'C#')
print(techno)
['C#', 'Python', 'java', 'SQL', 'C++']
techno.insert(2, 'R')
print(techno)
['C#', 'Python', 'R', 'java', 'SQL', 'C++']
```

- `pop()` – usuwa ostatni element z listy i zwraca do konsoli:

```
techno.pop()
'C++'
print(techno)
['C#', 'Python', 'R', 'java', 'SQL']
techno.pop()
'SQL'
```

```
print(techno)
```

```
['C#', 'Python', 'R', 'java']
```

- `pop(inx)` – usuwa element o indeksie `inx` z listy i zwraca do konsoli

```
techno = ['C#', 'Python', 'R', 'java', 'SQL']
```

```
techno.pop(0)
```

```
'C#'
```

```
print(techno)
```

```
['Python', 'R', 'java', 'SQL']
```

- `index()` – zwraca indeks przekazanej wartości do metody:

```
techno = ['C#', 'Python', 'R', 'java', 'SQL']
```

```
techno.index('R')
```

```
2
```

- `count()` – liczy liczbę wystąpień danego elementu:

```
techno = ['C#', 'python', 'R', 'R', 'java', 'SQL']
```

```
techno.count('R')
```

```
2
```

- `sort()` – sortowanie listy:

```
techno = ['c#', 'python', 'r', 'java', 'sql']
```

```
techno.sort()
```

```
print(techno)
```

```
['c#', 'java', 'python', 'r', 'sql']
```

```
techno.sort(reverse=True)
```

```
print(techno)
```

```
['sql', 'r', 'python', 'java', 'c#']
```

- odwracanie listy:

```
techno = ['c#', 'python', 'r', 'java', 'sql']
```

```
print(techno[::-1])
```

```
['sql', 'java', 'r', 'python', 'c#']
```

```
techno = ['c#', 'python', 'r', 'java', 'sql']
```

```
techno.reverse()
```

```
print(techno)
```

```
['sql', 'java', 'r', 'python', 'c#']
```

4.3.2 Krotki

- uporządkowana struktura, której po utworzeniu nie można aktualizować, są niezmiennie

- można z nich wycinać

1. utworzenie pustej krotki

```
pusta_krotka = tuple()
print(pusta_krotka)

()
```

2. krotki można definiować za pomocą nawiasów okrągłych:

```
amw = ('AMW', 'University', 'Poland', 1987)
print(amw)

('AMW', 'University', 'Poland', 1987)
```

3. przykłady obsługi krotek:

wybór i wydruk pierwszego elementu krotki:

```
print(amw[0])

AMW
```

próba zmiany krotki:

```
amw[0] = 'WSMW'

TypeError: 'tuple' object does not support item assignment
```

4. tworzenie krotek przez zagnieżdżanie:

```
wat = ('WAT', 'University', 'Poland', 1951)
dane = (amw, wat)
print(dane)

(('AMW', 'University', 'Poland', 1987), ('WAT', 'University', 'Poland', 1951))
```

5. kolejne zagnieżdżanie:

```
krotka_w_krotce = 'Paryż', 'Londyn', 'Rzym', ('Warszawa', 'Łódź')
print(krotka_w_krotce)

('Paryż', 'Londyn', 'Rzym', ('Warszawa', 'Łódź'))
```

6. przypisanie wartości zmiennym poprzez rozpakowanie krotki (wskazany restart jądra):

```
imie, nazwisko = ('Michał', 'Żółtko')
print(imie, nazwisko)
```

Michał Żółtko

```
skrot, sektor, kraj, rok_powstania = amw
(sprawdzić rozpakowanie krotki amw w zmiennych Spydera)
```

7. inny rodzaj definiowania krotki (bez nawiasów okrągłych):

```
uczelnie_wojskowe = 'AMW', 'WAT', 'ASW', 'AWL', 'LAW'
print(uczelnie_wojskowe)

('AMW', 'WAT', 'ASW', 'AWL', 'LAW')
```


8. ważne: bezpośrednia zamiana wartości zmiennych:

```
x, y = 10, 20
print(x, y)
10 20

x, y = y, x
print(x, y)
20 10
```

4.3.3 Słowniki

- nieuporządkowana struktura, zawiera pary klucz-wartość
- klucze nie mogą się powtarzać
- uporządkowanie nie jest istotne
- realizuje funkcję mapowania

1. tworzenie pustego słownika:

```
pusty_slownik = dict()
print(pusty_slownik)
{}

drugi_pusty_slownik = {}
print(type(drugi_pusty_slownik))
<class 'dict'>
```

2. tworzenie niepustego słownika:

```
pol_to_ang = {'jeden':'one', 'dwa':'two', 'trzy':'three'}
print(pol_to_ang)
{'jeden': 'one', 'dwa': 'two', 'trzy': 'three'}

name_to_digit = {'jeden':1, 'dwa':2, 'trzy':3}
print(name_to_digit)
{'jeden': 1, 'dwa': 2, 'trzy': 3}

digit_to_digit = {0:1, 1:2, 2:3}
print(digit_to_digit)
{0: 1, 1: 2, 2: 3}
```

3. sprawdzenie długości słownika (jeden element słownika to para klucz-wartość):

```
print( len(digit_to_digit) )
3
```

4. wstawianie elementów do słownika:

```
pol_to_ang['cztery'] = 'four'
```

```
print(pol_to_ang)
{'jeden': 'one', 'dwa': 'two', 'trzy': 'three', 'cztery': 'four'}
```

5. wydobywanie wartości ze słownika odbywa się za pomocą klucza:

6. popularne metody stosowane przy obsłudze słowników:

- `clear()` – czyści słownik:

```
pol_to_ang.clear()
print(len(pol_to_ang))
0
```

- `copy()` – kopiowanie słownika do innej zmiennej:

```
pol_to_ang = {'jeden': 'one', 'dwa': 'two', 'trzy': 'three'}
pol_to_eng_copied = pol_to_eng.copy()
```

7. `keys()` – metoda zwraca wszystkie klucze słownika (można je łatwo przekonwertować na listę):

```
pol_to_ang.keys()
dict_keys(['jeden', 'dwa', 'trzy'])
list(pol_to_ang.keys())
['jeden', 'dwa', 'trzy']
```

8. `values()` – zwraca wartości słownika (można je łatwo przekonwertować na listę):

```
pol_to_ang.values()
dict_values(['one', 'two', 'three'])
list(pol_to_ang.values())
['one', 'two', 'three']
```

9. `items()` – wydobywanie par klucz-wartość (można przekonwertować na listę krotek):

```
pol_to_ang.items()
dict_items([('jeden', 'one'), ('dwa', 'two'), ('trzy', 'three')])
list(pol_to_ang.items())
[('jeden', 'one'), ('dwa', 'two'), ('trzy', 'three')]
```

10. wydobywanie wartości ze słownika:

- w sposób bezpośredni za pomocą klucza, jeśli nie ma klucza zwraca błąd:

```
pol_to_ang['jeden']
'one'
pol_to_ang['zero']
KeyError: 'zero'
```

- `get()` – za pomocą metody, jeśli nie ma klucza to zwraca wartość domyślną

```
pol_to_ang.get('jeden')
'one'
```

```
pol_to_ang.get('zero', 'NaN')
'NaN'
```

11. `pop()` – usuwa element ze słownika i zwraca go do konsoli, wymagany jest klucz jako argument:

```
pol_to_ang.pop('dwa')
'two'
```

12. `popitem()` – zwraca i usuwa dowolny element słownika (parę: klucz-wartość):

```
pol_to_ang.popitem()
('trzy', 'three')
```

13. `update()` – zmienia wartość podanego klucza:

```
pol_to_ang = {'jeden': 'one', 'dwa': 'two', 'trzy': 'three'}
pol_to_ang.update({'jeden': 1})
```

4.3.4 Zbiory

- nieuporządkowany ciąg niepowtarzalnych elementów
- analogia do zbiorów matematycznych
- nie są zbyt często wykorzystywane w Pythonie

1. definiowanie zbioru pustego:

```
empty_set = set()
print(empty_set)
print(type(empty_set))
set()
<class 'set'>
```

2. przypisanie zbioru do zmiennej:

```
zbior = {'Python', 'c++', 'java', 'php'}
zbior
{'c++', 'java', 'php', 'Python'}
print(zbior)
{'Python', 'java', 'c++', 'php'}
print(type(zbior))
<class 'set'>
```

3. obliczenie długości zbioru (liczba elementów zbioru):

```
print(len(zbior))
4
```

4. działanie funkcji set na łańcuch:

```
set('Python')
{'P', 'h', 'n', 'o', 't', 'y'}
set('alama12kotów')
{'1', '2', 'a', 'k', 'l', 'm', 'o', 't', 'w', 'ó'}
```

5. sprawdzenie czy dane wartości znajdują się w zbiorze:

```
'Python' in zbior
```

```
True
```

```
'sql' in zbior
```

```
False
```

6. metody dostępne dla typu zbiorowego:

```
print(dir(set))
['__and__', '__class__', '__contains__', '__delattr__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__gt__', '__hash__', '__iand__',
 '__init__', '__init_subclass__', '__ior__', '__isub__',
 '__iter__', '__ixor__', '__le__', '__len__', '__lt__',
 '__ne__', '__new__', '__or__', '__rand__', '__reduce__',
 '__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__',
 '__setattr__', '__sizeof__', '__str__', '__sub__',
 '__subclasshook__', '__xor__', 'add', 'clear', 'copy',
 'difference', 'difference_update', 'discard', 'intersection',
 'intersection_update', 'isdisjoint', 'issubset', 'issuperset',
 'pop', 'remove', 'symmetric_difference',
 'symmetric_difference_update', 'union', 'update']
```

7. Metody dodawania i usuwania elementów do/ze zbioru

```
zbior.add('C#')
```

```
print(zbior)
```

```
{'C#', 'c++', 'php', 'java', 'Python'}
```

zwróć uwagę na losowe umiejscowienie nowego elementu

```
zbior.remove('C#')
```

```
{'c++', 'php', 'java', 'Python'}
```

usunięcie dowolnego element ze zbioru:

```
zbior.pop()
```

```
C++
```

```
print(zbior)
{'php', 'java', 'Python'}
```

8. Wyczyszczenie zbioru z elementów:

```
zbior.clear()
print(zbior)
set()
```

9. Sprawdzanie podzbioru i nadzbioru, suma, przecięcie i różnica symetryczna zbiorów

```
A = {1, 2, 3, 4, 5, 6, 7}
B = {5, 6, 7, 8, 9}
C = {6, 7}
C.issubset(A)
True
C.issubset({5, 7})
False
A.issuperset(C)
True
A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8, 9}
A.intersection(B)
{5, 6, 7}
A.symmetric_difference(B)
{1, 2, 3, 4, 8, 9}
```

10. Skopiowanie zbioru do innej zmiennej:

```
D = A.copy()
print(D)
{1, 2, 3, 4, 5, 6, 7}
```

4.4 Kontrola przepływu programu

Podczas implementowania algorytmów zachodzi konieczność podejmowania decyzji lub wykonywania pewnych kroków wielokrotnie. Przykładowo może być konieczne odrzucenie wartości, która nie pasuje do reszty danych, co wymaga podjęcia decyzji, albo w celu osiągnięcia pożądanego wyniku, na przykład podczas filtrowania danych, może zaistnieć potrzeba przetworzenia danych więcej niż raz. Python zaspokaja te potrzeby, udostępnia specjalne instrukcje, które pozwalają podejmować decyzje lub umożliwiają wykonywanie operacji więcej niż raz.

- Interpretacja Pythona wartości logicznych dla obiektów wybranych klas:

```
bool('')          -> False
bool(' ')         -> True
bool(0)           -> False
bool(1)           -> True
bool(0.0)         -> False
bool('0.0')       -> True
bool(set())       -> False
bool(list())      -> False
bool(tuple())     -> False
bool({'key': 'val'}) -> True
bool(['', ''])    -> True
```

4.4.1 Instrukcja warunkowa if

Instrukcji **if** (jeżeli) używamy do podejmowania decyzji. Przykład kodu:

```
def TestValue(zm_value):
    if zm_value == 5:
        print('Wartość zm_value wynosi 5')
    elif zm_value == 6:
        print('Wartość zm_value wynosi 6')
    else:
        print('Wartość zm_value jest inna.')
        print('Wynosi ' + str(zm_value))
```

- Każda instrukcja **if** w Pythonie zaczyna się od słowa **if**.
- Oznacza to dla Pythona, że trzeba podjąć decyzję.
- Po słowie **if** występuje warunek, który określa rodzaj porównania, jakie ma dokonać Python.
- W tym przykładzie powyżej jest to ustalenie, czy wartość parametru `Value` wynosi 5.
- W warunku użyto relacyjnego operatora równości `==`, a nie operatora przypisania `=`!!
- Warunek zawsze kończy się dwukropkiem `:`.
- Za dwukropkiem należy wymienić zadania, które ma wykonać Python.
- W pojedynczej instrukcji **if** może być konieczne wykonanie wielu zadań.
- Klauzula **elif** umożliwia wprowadzenie dodatkowego warunku i powiązanych z nim zadań.
- Klauzula ta jest dodatkiem do poprzedniego warunku, który w tym przypadku był podany w instrukcji **if**.

- W klauzuli **elif**, podobnie jak w instrukcji **if**, zawsze występuje warunek.
- Z klauzulą **elif** jest także związany odrębny zestaw zadań do wykonania.
- Ponadto jeżeli warunki **if** oraz **elif** nie zapewniają wszystkich możliwości wyboru decyzji, można całą instrukcję **if** zakończyć klauzulą **else**.
- Klauzule ta mówi Pythonowi, aby zrobił coś konkretnego w sytuacji, gdy warunki wcześniejsze nie są spełnione.
- **Zagnieżdżanie** to proces umieszczania instrukcji podrzędnej w innej instrukcji. W większości przypadków można zagnieżdżać dowolną instrukcję w dowolnej innej instrukcji. Oto przykład zagnieżdżania instrukcji **if**.

```
def SecretNumber():
    One = int(input("Wpisz pierwszą liczbę od 1 do 10: "))
    Two = int(input("Wpisz drugą liczbę od 1 do 10: "))

    if One >= 1 and One <=10:
        if Two >= 1 and Two <= 10:
            print('Twoja tajna liczba to: ' + str(One * Two))
        else:
            print('Niepoprawna druga liczba!')
    else:
        print('Niepoprawna pierwsza liczba!')
```

- Funkcja `SecretNumber()` prosi użytkownika podanie dwóch liczb (`input()`).
- Funkcja `int()` konwertuje wprowadzone dane wejściowe na liczby.
- Występują dwa poziomy instrukcji **if**: pierwszy sprawdza poprawność liczby zapisanej w zmiennej `One`, a drugi poziom sprawdza poprawność liczby zapisanej w zmiennej `Two`.
- Gdy zarówno zmienna `One`, jak i zmienna `Two` mają wartości od 1 do 10, funkcja `SecretNumber()` wyświetla użytkownikowi tajną liczbę.
- Aby zobaczyć działanie funkcji `SecretNumber()` wpisz polecenie `SecretNumber()` i środowisku Pythona.
- Wybrane zagadnienia związane z instrukcją **if**:
 1. sprawdzanie wartości logicznej łańcucha znaków:


```
string = ''
if string:
    print('Niepusty ciąg znaków')
else:
    print('Pusty ciąg znaków')
```
 2. sprawdzanie wartości logicznej liczby:


```
numer = 0
```

```

if numer:
    print('Liczba niezerowa')
else:
    print('Zero')

```

3. sprawdzanie wartości logicznej określonej flagi:

```

flaga = True
if flaga == True:
    print('Doszło do zdarzenia')
else:
    print('Nie było zdarzenia')

```

uwaga: fragment `== True` jest zbędny

4. testowanie istnienia znaku w łańcuchu znakowym:

```

name = 'Python'
if 'p' in name:
    print('Znaleziono p')
else:
    print('Nie znaleziono p')

```

5. **Ważne:** Zapis instrukcji **if** w jednej linii:

- najpierw przykład **if** w wersji standardowej:

```

techno = 'Python'
if techno == 'Python':
    flaga = 'Dobry wybór'
else:
    flaga = 'Wybierz Pythona'

```

- teraz **if** w jednej linii:

szablon:

```

x if [warunek] else y

```

```

techno = 'Python'
flaga = 'Dobry wybór' if techno == 'Python' else 'Wybierz Pythona'
print(flaga)
'Dobry wybór'

```

4.4.2 Pętla for

- Jeżeli trzeba wykonać zadanie większą liczbę razy, ale konkretnie wiadomo ile, to można skorzystać z pętli zorganizowanej przez instrukcję **for**.
- Pętla **for** ma określony początek i określony koniec.
- Liczba iteracji tej pętli zależy od liczby elementów w podanej zmiennej.

- Można iterować po wielu różnych elementach: łańcuchach znakowych, listach, krotkach, itd.
- Aby zobaczyć jak to działa zapoznaj się lub wykonaj poniższe przykłady:

1. Prosty przykład:

```
for znak in 'Python':
    print(znak)
p
y
t
h
o
n
```

lub z modyfikacją:

```
nazwa = 'Python'
index = 0
for znak in nazwa:
    print(index, znak)
    index += 1
0 p
1 y
2 t
3 h
4 o
5 n
```

2. pętla **for** z funkcją `enumerate()`:

- funkcja `enumerate()` zwraca krotki indeksów i wartości zmiennej iterowanej:

```
for index in enumerate(nazwa):
    print(index)
(0, 'p')
(1, 'y')
(2, 't')
(3, 'h')
(4, 'o')
(5, 'n')
```

- zwracane krotki z funkcji `enumerate` można rozpakować i zmienne wykorzystać:

```
for index, znak in enumerate(nazwa):
    print(index, znak)
0 p
1 y
2 t
3 h
4 o
5 n
```

3. Ważne: funkcja `enumerate` jest zalecanym sposobem iterowania po obiektach iterowalnych, oto kolejny przykład:

```

for i, wartosc in enumerate([1, 2, 3, 4, 5, 4]):
    print(i, wartosc)
0 1
1 2
2 3
3 4
4 5
5 4

```

4. pętla **for** z funkcją `range()`:

- funkcja `range()` zwraca obiekt iterowalny:

```
range(5)
```

```
range(0, 5)
```

```
range(len(nazwa))
```

```
range(0, 6)
```

- ze zwróconego obiektu typu `range` można łatwo zrobić listę:

```
list(range(5))
```

```
[0, 1, 2, 3, 4]
```

- połączenie **for** i `range`:

```
for index in range(5):
```

```
    print(index)
```

```
0
```

```
1
```

```
2
```

```
3
```

```
4
```

5. połączenie **for**, `range` i `len`:

```
for index in range(len(nazwa)):
```

```
    print('Nr indesu: ', index, 'Litera: ', nazwa[index])
```

```
Nr indesu: 0 Litera: p
```

```
Nr indesu: 1 Litera: y
```

```
Nr indesu: 2 Litera: t
```

```
Nr indesu: 3 Litera: h
```

```
Nr indesu: 4 Litera: o
```

```
Nr indesu: 5 Litera: n
```

6. przykłady wykorzystujące funkcję `range()`:

```
for i in range(10, 15):
```

```
    print(i)
```

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

```

for i in range(10, 20, 2):
    print(i)
10
12
14
16
18
for i in range(5, 0, -1):
    print(i)
5
4
3
2
1

```

7. przykład pętli **for** z łańcuchami znakowymi:

```

tekst = 'Nauka Pythona'
for znak in tekst[:5]:
    print(znak)
N
a
u
k
a

```

8. pętla **for** z funkcją `zip()`:

```

for numer, znak in zip('123456', 'Python'):
    print(numer, znak)
1 P
2 y
3 t
4 h
5 o
6 n

```

9. instrukcja **break** – zatrzymuje wykonywanie pętli:

```

for i in '0123456789':
    i = int(i)
    print(i)
    if i == 6:
        break
print('Koniec')
0
1
2
3
4
5
6
Koniec

```

```

tekst = 'Nauka Pythona'
for znak in tekst:
    if znak == ' ':
        break
    print(znak)
print('Koniec')

```

N
a
u
k
a
Koniec

10. pętla **for** z klauzulą **else**:

```

for znak in 'jannowak@gmail.com':
    if znak == '@':
        print('Adres email poprawny')
        break

```

Adres email poprawny

```

for znak in 'jannowak@gmail.com':
    if znak == '@':
        print('Adres email poprawny')
        break
else:
    print('Adres email nie jest poprawny')

print('Koniec')

```

Adres email poprawny
Koniec

```

for znak in 'jannowakgmail.com':
    if znak == '@':
        print('Adres email poprawny')
        break
else:
    print('Adres email nie jest poprawny')

print('Koniec')

```

Adres email nie jest poprawny
Koniec

11. **continue** – wymusza przejście pętli od bieżącego punktu wykonania do następnej pozycji (nie opuszcza pętli tylko pomija część instrukcji):

```

for i in range(10):
    if i % 2 == 1:
        continue
    print(i)

```

0
2

4
6
8

```
tekst = 'Nauka Pythona'
for znak in tekst:
    if znak == ' ':
        continue
    print(znak)
```

N
a
u
k
a
P
y
t
h
o
n
a

```
def DisplayMulti(*varArgs):
    for arg in varArgs:
        if arg.upper() == 'CONT':
            print('Argument kontynuacji: ' + arg)
            continue
        elif arg.upper() == 'BREAK':
            print('Argument przzerwania pętli: ' + arg)
            break
        print('Dobry argument: ' + arg)
```

- Powyższa pętla **for** próbuje przetworzyć wszystkie elementy w VarArgs. W pętli znajduje się zagnieżdżona instrukcja **if**, która sprawdza dwa warunki brzegowe.
- W większości przypadków kod pomija instrukcję **if** i po prostu wyświetla argument.
- Jednak jeśli instrukcja **if** znajdzie słowa CONT lub BREAK we wprowadzonych wartościach wejściowych, wykonuje jedno z następujących dwóch zadań:

continue – wymusza przejście pętli od bieżącego punktu wykonania do następnej pozycji w VarArgs.

Słowa kluczowe mogą być wprowadzane przy użyciu dowolnej kombinacji wielkich i małych liter, np. ConT, ponieważ funkcja upper() dokonuje konwersji ciągu na wersję zapisaną wielkimi literami. Funkcja DisplayMulti() może przetwarzać dowolną liczbę wprowadzonych ciągów znaków. Aby

zobaczyć działanie tej funkcji wpisz:

```
DisplayMulti('Witaj', 'Żegnaj', 'Pierwszy', 'Ostatni')
```

Następnie wywołaj funkcję wpisując:

```
DisplayMulti('Witaj', 'Cont', 'Żegnaj', 'Pierwszy', 'Break', 'Ostatni')
```

4.4.3 Pętla while

Instrukcja pętli **while** wykonuje zadania do czasu spełnienia warunku. Podobnie jak **for** obsługuje słowa kluczowe **continue** oraz **break**, które pozwalają na przedwczesne zakończenie pętli. Aby zobaczyć jak to działa wykonaj poniższy kod:

```
def SecretNumber():
    GotIt = False
    while GotIt == False:
        One = int(input("Wpisz liczbę od 1 do 10: "))
        Two = int(input("Wpisz liczbę od 1 do 10: "))

        if (One >= 1) and (One <=10):
            if (Two >= 1) and (Two <= 10):
                print('Twoja tajna liczba to: ' + str(One * Two))
                GotIt = True
                continue
            else:
                print('Niepoprawna druga liczba!')
        else:
            print('Niepoprawna pierwsza liczba!')
    print("Spróbuj jeszcze raz!")
```

Jest to rozwinięcie poprzedniej funkcji `SecretNumber()`, ale z powodu dodania instrukcji pętli `while`, funkcja będzie kontynuować zadawanie pytań o wprowadzenie danych wejściowych, dopóki nie otrzyma prawidłowej odpowiedzi.

Aby zobaczyć jak działa powyższa funkcja, wpisz `SecretNnumber()` w środowisku Pythona.

Wprowadź dane: 20 i 10 – dostaniesz monit, że pierwsza liczba jest błędna i ponowna zachętę.

Wprowadź dane: 10 i 20 – dostaniesz monit, że druga liczba jest niepoprawna oraz ponowna zachętę. Wprowadź teraz liczby: 10 i 10. Teraz program wyliczy tajny numer, wyświetli go, a ponieważ zadziała klauzula **continue** aplikacja nie poprosi o ponownie próby.

4.5 Funkcje

4.5.1 Przegląd wbudowanych funkcji Pythona

1. `abs()` – zwraca wartość bezwzględną liczby całkowitej lub zmiennoprzecinkowej:

```
print(abs(10), abs(-10.5))  
10 10.5
```

2. `bool()` – test logiczny argumentu, funkcja zwraca jedną z dwóch wartości `True` lub `False`:

```
bool([])           -> False  
bool('')          -> False  
bool({})          -> False  
bool(' ')         -> True  
bool(True)        -> True  
bool(False)       -> False  
bool(10)          -> True  
bool(0)           -> False
```

3. `dir()` – zwraca wszystkie metody i atrybuty danego obiektu:

```
dir(list)  
dir(tuple)
```

4. `enumerate()` – zwraca obiekt `enumerate`, który tworzy listę krotek; jest wykorzystywana do iterowania i jest często wykorzystywana z listą:

```
list(enumerate(['Python', ' ', 'java', 'c++']))  
[(0, 'Python'), (1, ' '), (2, 'java'), (3, 'c++')]
```

5. `eval()` – pozwala wykonać działania, które są przekazywane jako tekst:

```
eval('10 + 5')  
15  
x = 100  
eval('x + 22')  
122
```

6. `filter()` – pozwala filtrować dane, zwraca tylko te wartości, dla których wartość logiczna jest `True`

```
list(filter(abs, [-2, -1, 0, 1, 2]))  
[-2, -1, 1, 2]  
  
list(filter(bool, ['Python', ' ', 'java']))  
['Python', 'java']
```

7. `float()` – konwersja liczby lub liczby w postaci tekstu na liczbę zmiennoprzecinkową:

```
float(10)           -> 10.0  
float('10')        -> 10.0
```

```
float(10.5)      -> 10.5
float('10.5')   -> 10.5
float('abc')     -> błąd
```

8. `type()` – zwraca typ wartości argumentu:

```
type(10)         -> int
type('abc')      -> str
```

9. `help()` – wydruk pomocy na temat innego elementu

```
help(type)
opis ...
```

10. `isinstance()` – argumentem jest object, sprawdza czy obiekt przynależy do danej klasy:

```
isinstance(1, int)      -> True
isinstance(1, float)    -> False
isinstance('abc', str)  -> True
isinstance('', str)     -> True
```

11. `len()` – zwraca długość danego argumentu:

```
len('Python')          -> 6
len('')                 -> 0
len(' ')                -> 1
len([])                 -> 0
len([1, 2], [3, 4, 5, 6, [7, 8]]) -> 2
```

12. `list()` – przyjmuje obiekt iterowalny i tworzy z niego listę:

```
list('Python')          -> ['p', 'y', 't', 'h', 'o', 'n']
list(range(10))          -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

13. `map()` – podobnie jak `filter`, pobiera funkcję i obiekt iterowalny; aby ją podglądnąć należy użyć `list()`

```
list(map(abs, [-2, -1, 0, 1, 2])) -> [2, 1, 0, 1, 2]

nazwy = ['Python', 'java', 'c++']
list(map(str.title, nazwy))      -> ['Python', 'Java', 'C++']
```

14. `min()` i `max()` – zwracają wartości minimalne i maksymalne elementu z obiektu iterowalnego:

```
min([1, 2, 30, 4, 5]) -> 1
max([1, 2, 30, 4, 5]) -> 30
min('Python')          -> 'h'
```



```
max('Python') -> 'y'
```

15. `pow()` – podnosi do potęgi

```
pow(10, 2) -> 100
```

16. `reversed()` – odwraca kolejność obiektu iterowalnego:

```
list(reversed([10, 20, 50, 60])) -> [60, 50, 20, 10]  
list(reversed('Python')) -> ['n', 'o', 'h', 't', 'y', 'p']
```

17. `round()` – zaokrągla wartości do określonego miejsca po przecinku:

```
round(5.6789) -> 6  
round(5.6789, 2) -> 5.68
```

18. `str()` – przekształca liczbę na tekst:

```
str(10) -> '10'  
str(123.456) -> '123.456'
```

19. `sum()` – suma elementów:

```
sum([1, 2, 3, 4, 5]) -> 15
```

20. `zip()` – iterowanie po kilku elementach iterowalnych jednocześnie; zwraca krotki; ogranicza ją najmniejszy element:

```
lista_1 = [1, 2,]  
lista_2 = [4, 5, 6]  
lista_3 = [7, 8, 9, 0]  
list(zip(lista_1, lista_2, lista_3))  
[(1, 4, 7), (2, 5, 8)]  
  
list(zip('Python', 'kurs'))  
[('p', 'k'), ('y', 'u'), ('t', 'r'), ('h', 's')]
```

4.5.2 Definiowanie własnych funkcji

- funkcje definiuje się przez słówko `def`
- funkcje wywołuje się przez podanie jej nazwy z nawiasami okrągłymi
- funkcja może zwracać wartości
- ponieważ w Pythonie wszystko jest obiektem, definiując funkcję tworzymy obiekt klasy `function`, który jest zaimplementowany natywnie w Pythonie

1.

```
def fun_0():
    print('Działa funkcja')
fun_0()
print(type(fun_0))
```

Działa funkcja
<class 'function'>

2. przykład prostej funkcji bez argumentów (tekst funkcji wpisany do skryptu i skrypt jest uruchomiony)

```
def fun_1():
    print('Uruchomiono funkcję fun_1.')

fun_1()
```

Uruchomiono funkcję fun_1.

3. przykład funkcji z argumentami

```
def fun_2(x, y):
    print('Argumenty funkcji to: {}, {}'.format(x, y))

fun_2(10, 20)
```

Argumenty funkcji to: 10, 20

4. przykłady funkcji z argumentami domyślnymi

```
# drugi argument jest domyślny, w wywołaniu muszą być podane dwa argumenty
def fun_2(x, y = 50):
    print('Argumenty funkcji to: {}, {}'.format(x, y))

fun_2(10, 20)
```

Argumenty funkcji to: 10, 20

oba argumenty mają wartości domyślne, wywołanie funkcji może być bez argumentów

```
def fun_2(x = 10, y = 50):
    print('Argumenty funkcji to: {}, {}'.format(x, y))

fun_2()
```

Argumenty funkcji to: 10, 50

w wywołaniu funkcji można zmieniać kolejność argumentów, podając ich nazwy

```
def fun_2(x = 10, y = 50):
    print('Argumenty funkcji to: {}, {}'.format(x, y))
```

```
fun_2(y = 100, x =200)
```

Argumenty funkcji to: 200, 100

5. funkcja zwracająca wartości:

```
def dodaj(x, y):  
    return x + y
```

```
wynik = dodaj(5, 10)  
print(wynik)  
15
```

5 Przykładowe zbiory danych

W trakcie omawiania zagadnień prezentowanych na zajęciach wykorzystujemy różne zbiory danych, które są dostępne w zasobach bibliotek Pythona lub można je pobrać z określonych lokalizacji Internetu. Te przykładowe zbiory danych mają wspomagać zrozumienie, opanowanie i zapamiętanie materiału. Poniżej zamieszczono przegląd najczęściej używanych zbiorów.

5.1 Przykładowe zbiory danych dostępne w ogólnodostępnych zasobach oraz w pakiecie `scikit-learn`

W pakiecie `scikit-learn` znajduje się kilka zbiorów danych, które można łatwo pobrać za pomocą polecenia `import`. Nie ma więc potrzeby sięgania do zewnętrznych źródeł i repozytoriów internetowych. Takimi zbiorami, które są bardzo często wykorzystywane na potrzeby klasyfikacji i regresji są *Iris*, *Digits* i *Boston*. Zbiory te mają postać obiektów podobnych do słownika i oprócz cech oraz zmiennych docelowych obejmują kompletny opis oraz objaśnienie kontekstu danych.

5.1.1 Zbiór danych *Iris*







1. https://en.wikipedia.org/wiki/Iris_flower_data_set
2. Został utworzony w 1936 roku przez Ronalda Fishera (zajmował się analizą statystyczną) w celu zilustrowania liniowej analizy dyskryminacyjnej przeprowadzanej na małym zbiorze empirycznie weryfikowalnych przykładów, gdzie każda ze 150 obserwacji reprezentuje jeden z 3 gatunków irysa: *Setosa*, *Versicolor*, *Virginica*, po 50 przykładów każdego gatunku. Dane obejmują cztery zmienne opisowe (cechy), które wspólnie pozwalają podzielić zbiór na klasy.
3. Zaletą tego zbioru jest łatwość jego wczytywania, obsługi i eksploracji celach uczenia nadzorowanego lub generowania reprezentacji graficznych, co wynika z małej liczby wymiarów w zbiorze danych, a tworzenie modeli przy wykorzystaniu tego zbioru trwa bardzo krótko i nie jest wymagana skomplikowana konfiguracja sprzętowa komputera. Ponadto rola zmiennych objaśniających (cech) oraz relacje między klasami są dobrze znane.
4. Ponadto biblioteka `pandas` dostarcza wygodną strukturę `DataFrame`, która umożliwia łatwą obsługę zbiorów danych w postaci macierzy (wiersze i kolumny) obejmujące zmienne różnych typów.
5. W celu wczytania zbioru danych *Iris* z pakietu `scikit-learn` należy wydać polecenie:

```
from sklearn import datasets
iris = datasets.load_iris()
```

```
print(type(iris))
print(type(iris.data))
```

6. Po wczytaniu danych można zapoznać się z ich opisem i ustalić sposób przechowywania cech i wartości docelowych.



7. Wszystkie zbiory danych z pakietu `scikit-learn` udostępniają metody:

-  `.DESCR` – zwraca ogólny opis zbioru danych
-  `.data` – zwraca wszystkie cechy
-  `.feature_names` – zwraca nazwy cech
-  `.target` – zwraca docelowe wartości w postaci konkretnych wartości lub numerów klas
-  `.target_names` – zwraca nazwy klas w docelowych wartościach
-  `.shape` – używana do wyników metod `.data` i `.target`; jako pierwszą wartość zwraca liczbę obserwacji, a jako drugą – liczbę cech

8. A zatem, można się dowiedzieć więcej na temat zbioru danych, sprawdzić ile jest przykładów, zmiennych i jakie są ich nazwy. Zastosuj poniższe polecenia z instrukcją `print` i dokonaj analizy zbioru danych:

```
print(iris.DESCR)
print(iris.data)
print(iris.data.shape)
print(iris.feature_names)
print(iris.target)
print(iris.target.shape)
print(iris.target_names)
```

9. Należy zauważyć, że w zbiorze danych znajdują się dwie tablice:

-  tablica pierwsza: `data` służy do przechowywania wartości liczbowych czterech cech `sepal length`, `sepal width`, `petal length` i `petal width` uporządkowanych w macierz o wymiarach `150 x 4`, gdzie 150 to liczba obserwacji, a 4 to liczba cech.
-  tablica druga: `target` jest w istocie wektorem i służy do przechowywania wartości całkowitoliczbowych do reprezentowania konkretnej klasy, przy czym każda z tych wartości jest tak naprawdę indeksem dla nazwy klasy zawartej w `target_names`; np. wartość 0 w wektorze wartości docelowych `target` reprezentuje gatunek `setosa`.

10. W bibliotece `pandas` znajduje się specjalna funkcja do wygenerowania macierzy wykresów punktowych, która pozwala na obserwację zależności i rozkładów zmiennych ilościowych z danego zbioru. Wykonać poniższy kod:

```
import pandas as pd
import numpy as np
colors = list()
palette = {0: 'red', 1: 'green', 2: 'blue'}

for c in np.nditer(iris.target): colors.append(palette[int(c)])
dataframe = pd.DataFrame(iris.data, columns=iris.feature_names)
sc = pd.plotting.scatter_matrix(dataframe, alpha=0.3,
figsize=(10, 10), diagonal='hist', color=colors, marker='o',
grid=True)
```

11. Przykład wykorzystania zbioru danych *Iris* w postaci pliku jako zasób dostępny w Internecie.

Pobrać na dysk lokalny plik z danymi z poniższej strony i wykonać kod:

<http://archive.ics.uci.edu/ml/datasets/iris>

```
iris = pd.read_csv('dataset-uci-iris.csv', sep=',', decimal='.',
header=0, names=['sepal_lenght', 'sepal_width', 'petal_lenght',
'petal_width', 'target'])
print(type(iris))
```

Uwagi:

- w poleceniu `pd.read_csv` można podać separator i znak oddzielający część ułamkową (`decimal`), określić czy występuje nagłówek (`header=0` lub `header=None`) oraz podać nazwy zmiennych w postaci listy
- ponieważ w przykładzie zdefiniowano zmienne w postaci pojedynczych słów (stosowane są znaki podkreślenia) to można się odwoływać do nich w notacji z kropką, tzn. np. :

```
iris.sepal_lenght
```

- strukturę `DataFrame` obiektu *iris* utworzonego dzięki bibliotece `pandas` można przekształcić na zestaw wdców tablic pakietu `NumPy` zawierających obserwacje (cechy) i wartości docelowe (etykiety klas):

```
iris_data = iris.values[:, :4]
iris_target, iris_target_labels = pd.factorize(iris.target)
```

```
print(type(iris_data))  
print(iris_data)  
print(type(iris_target))  
print(iris_target)  
print(iris_data.shape, iris_target.shape)
```