Wprowadzenie do programowania

dr inż. Sławomir Koczubiej

Politechnika Świętokrzyska
Wydział Zarządzania i Modelowania Komputerowego
Katedra Technologii Informatycznych

(9 października 2025)

Kontakt

Budynek C, pokój 3.26
sk@tu.kielce.pl

Materiały do pobrania, aktualności, terminy zaliczeń

http://staff.tu.kielce.pl/sk

Organizacja wykładów

- Wykłady są nieobowiązkowe (zgodnie z postanowieniami regulaminu), ale...
- Na wykłady warto zaglądać.
- Czasem sprawdzam listę obecności.

Warunki zaliczenia wykładu

Test zaliczeniowy po zakończeniu wykładów.

Organizacja laboratoriów

- Zajęcia laboratoryjne są obowiązkowe.
- Dopuszcza się jedną nieobecność.
- Większa liczba nieobecności powoduje zmniejszenie oceny do niedostatecznej włącznie (3 lub więcej nieobecności).
- W przypadku usprawiedliwionej nieobecności zajęcia można odrobić z inną grupą (jeśli istnieje taka możliwość).

Warunki zaliczenia laboratoriów

Wykonanie ćwiczeń i zaliczenie sprawdzianów kontrolnych.

Treść wykładów

- Pojęcie algorytmu, języki zapisu algorytmów. Przykłady algorytmów.
- Paradygmaty programowania. Wprowadzenie do programowania, semantyka i syntaktyka języka programowania.
- Operatory, wyrażenia algebraiczne i logiczne.
- Instrukcje wejścia/wyjścia. Proces translacji oraz uruchamiania programu.
- Reprezentacja danych w pamięci komputera. Podstawowe typy danych: liczbowe, znakowe. Złożone typy danych.
- Instrukcje sterujące: warunkowa, wyboru, iteracyjne.
- Korzystanie z wbudowanych funkcji i bibliotek języka. Zapis programów z użyciem własnych podprogramów. Przekazywanie parametrów do podprogramów. Zasięg zmiennych.
- Typ plikowy. Obsługa różnego rodzaju plików (tekstowe, binarne).

Treść laboratoriów

- Struktura programu w języku programowania. Import bibliotek. Korzystanie z funkcji bibliotecznych.
- Operacje wejścia-wyjścia. Komunikacja z użytkownikiem. Formatowanie danych.
- Operatory w programie, definiowanie wyrażeń. Własności operatorów (priorytet, łączność).
- Instrukcje warunkowa i przełączająca.
- Kontenery, definiowanie kontenerów, przetwarzanie kontenerów, instrukcje pętli. Algorytmy przetwarzania iteracyjnego.
- Definiowanie podprogramów. Zasięg zmiennych. Parametry podprogramów i sposoby ich przekazywania.
- Programowanie z wykorzystaniem plików tekstowych i binarnych. Debugowanie i testowanie programu.

Literatura

- Dokumentacja poszczególnych narzędzi programistycznych.
- Wróblewski P. *Algorytmy, struktury danych i techniki programowania.* Wydawnictwo Helion, Gliwice 2015.
- Wirth N. *Algorytmy + struktury danych = programy.* Wydawnictwo WNT, Warszawa 2001.
- Lutz M. *Python. Wprowadzenie.* Wydawnictwo Helion, Gliwice 2020.
- Saha A. *Matematyka w Pythonie.* Wydawnictwo Helion, Gliwice 2021.
- Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. *Wprowadzenie do algorytmów.* Wydawnictwo WNT, Warszawa 2004.

Program komputerowy działa według określonego algorytmu.

Algorytm – ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadania.

Zapis algorytmu działania w wybranym języku programowania nazywamy **implementacją algorytmu**.

Jako przykład stosowanego w życiu codziennym algorytmu podaje się często przepis kulinarny. Dla przykładu, aby ugotować bigos należy w określonej kolejności oraz odstępach czasowych (imperatyw czasowy) dodawać właściwe rodzaje kapusty i innych składników.

Przykład ten ma wyłącznie charakter poglądowy, ponieważ język przepisów kulinarnych nie został jasno zdefiniowany. Algorytmy zwykle formułowane są w sposób ścisły w oparciu o język matematyki.

Algorytm prowadzi do rozwiązania zadania w **skończonej liczbie kroków**.

Do danego celu prowadzi zwykle więcej niż jedna droga. Jak więc oceniać alternatywne sposoby rozwiązania problemu?

Podstawowe parametry algorytmu to jego **złożoność czasowa** i **złożoność pamięciowa**. Oprócz tego przy algorytmach działających na liczbach trzeba pamiętać o **stabilności numerycznej**.

Złożoność czasowa mówi, ile kroków obliczeniowych i ile czasu wymaga zakończenie algorytmu dla danej porcji danych.

Złożoność pamięciowa mówi, jaką maksymalnie część danych i wyników pośrednich trzeba w ramach danego algorytmu przechowywać w pamięci operacyjnej.

Stabilność numeryczna określa wrażliwość wyniku końcowego na błędy zaokrągleń w trakcie obliczeń oraz na dokładność danych początkowych.

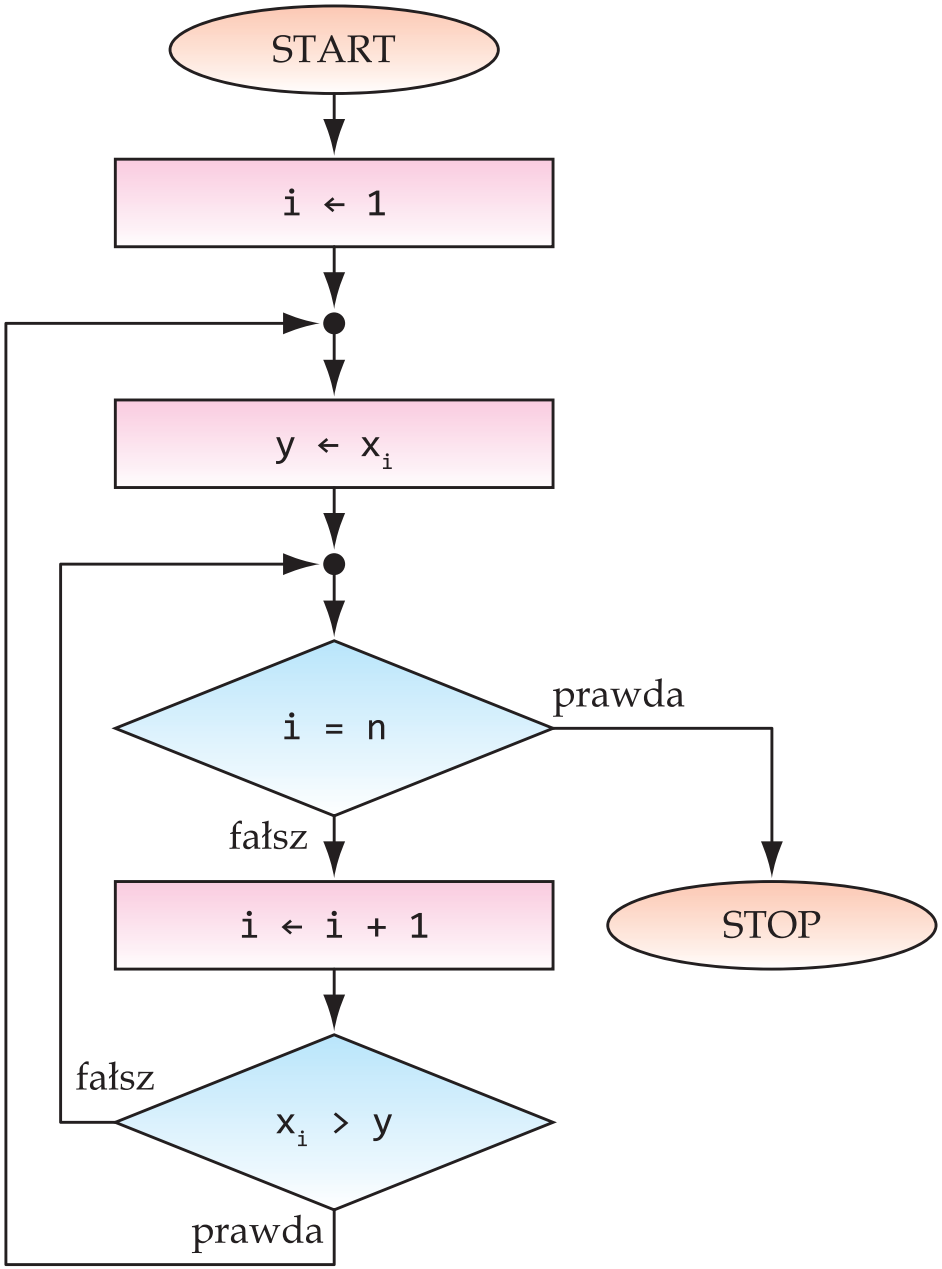
Sposoby zapisu algorytmów

Algorytm znajdowania wartości $y = \max\{x_i\}$, gdzie $1 \leq i \leq n$.

- **Język naturalny**.
- Schemat blokowy.
- Język formalny.

1. $i \leftarrow 1$, idź do 2,
2. $y \leftarrow x_i$, idź do 3,
3. Czy $i = n$? Jeśli tak – koniec, jeśli nie – idź do 4,
4. $i \leftarrow 1 + 1$, idź do 5,
5. Czy $x_i > y$? Jeśli tak – idź do 2, jeśli nie – idź do 3.

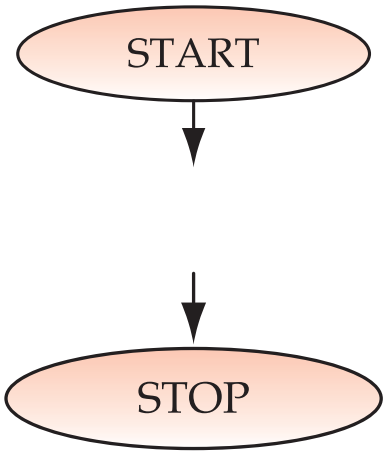
- Język naturalny.
- **Schemat blokowy**.
- Język formalny.



- Język naturalny.
- Schemat blokowy.
- **Język formalny**, C.

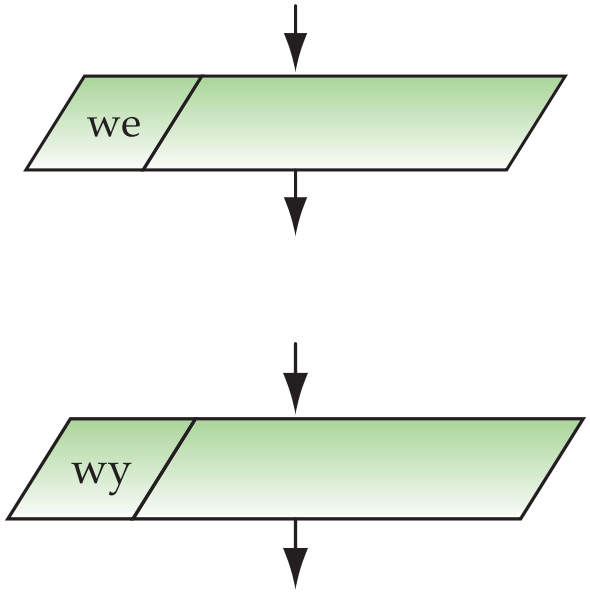
```c
int maxValue(int array[], int size)
{
  int i, max;

  max = array[0];

  for(i = 1; i < size; i++)
  {
    if(array[i] > max)

      max = array[i];
  }

  return max;
}
```

Schematy blokowe



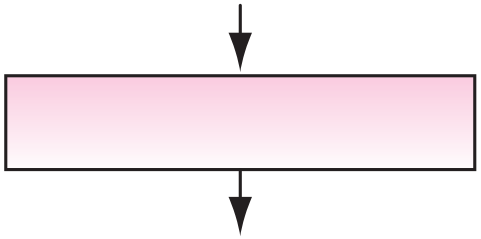
Start – początek programu

Stop – koniec programu

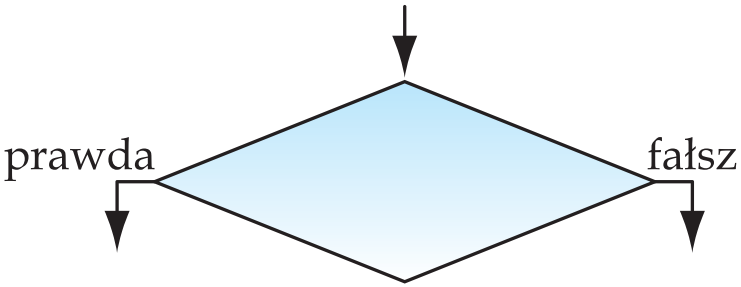


Blok wejścia – wprowadzanie danych

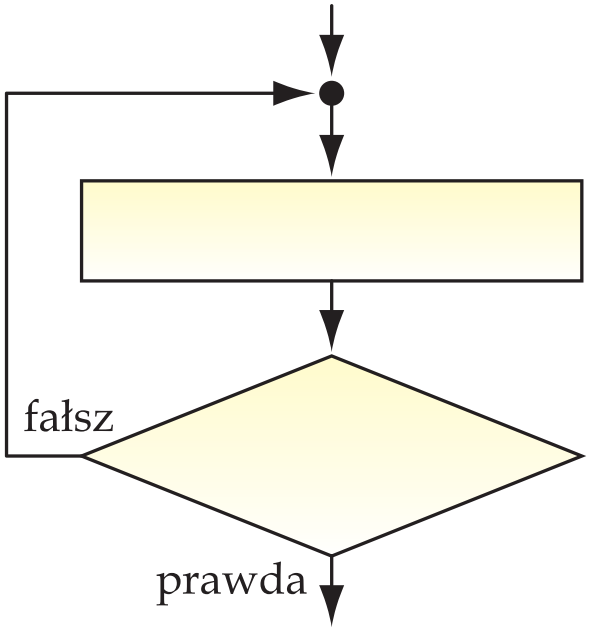
Blok wyjścia – wyprowadzanie wyników



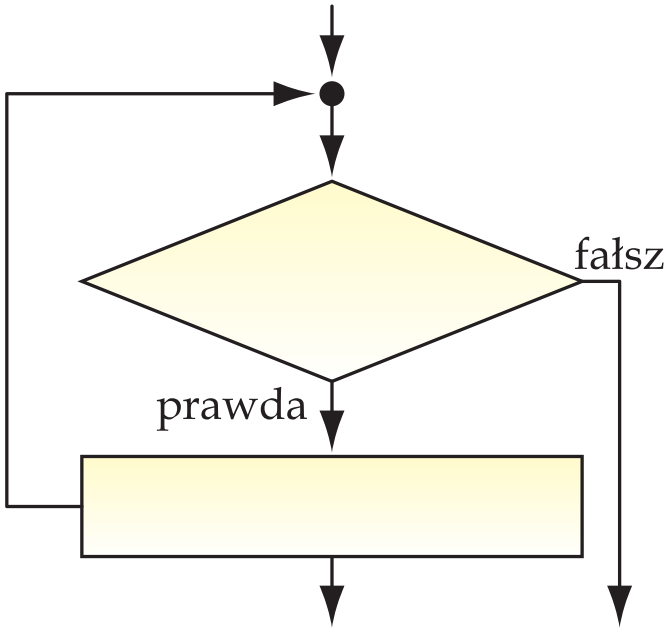
Blok operacyjny – przetwarzanie danych



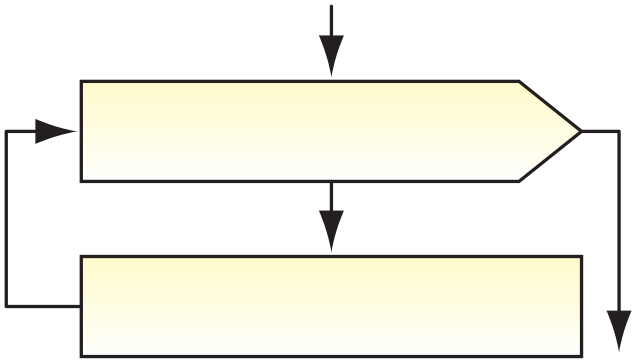
Blok decyzyjny – instrukcja warunkowa



Pętla *powtórz* – z warunkiem po bloku operacyjnym



Pętla *dopóki* – z warunkiem przed blokiem operacyjnym

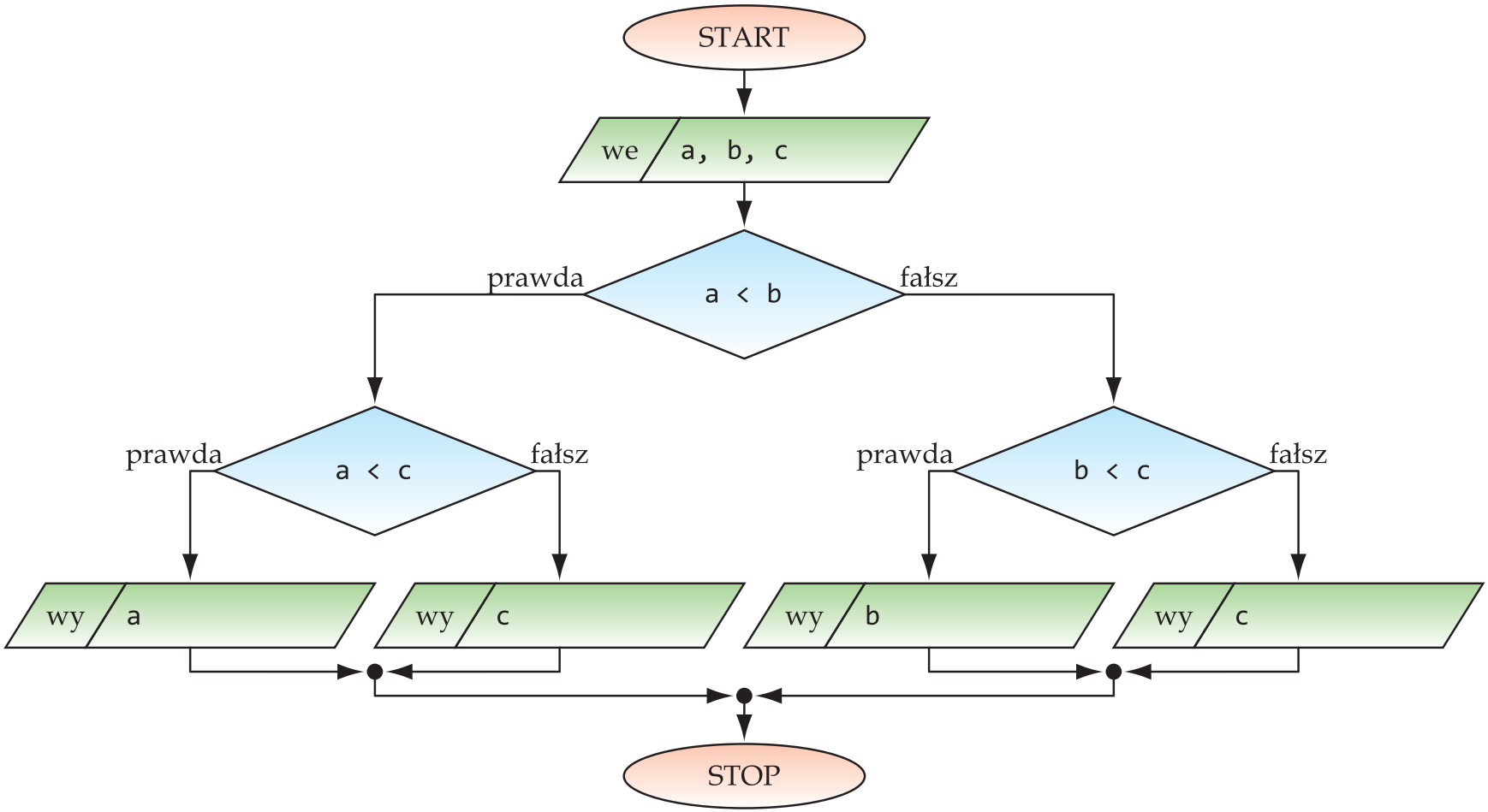


Pętla *dla* – z wiadomą liczbą iteracji

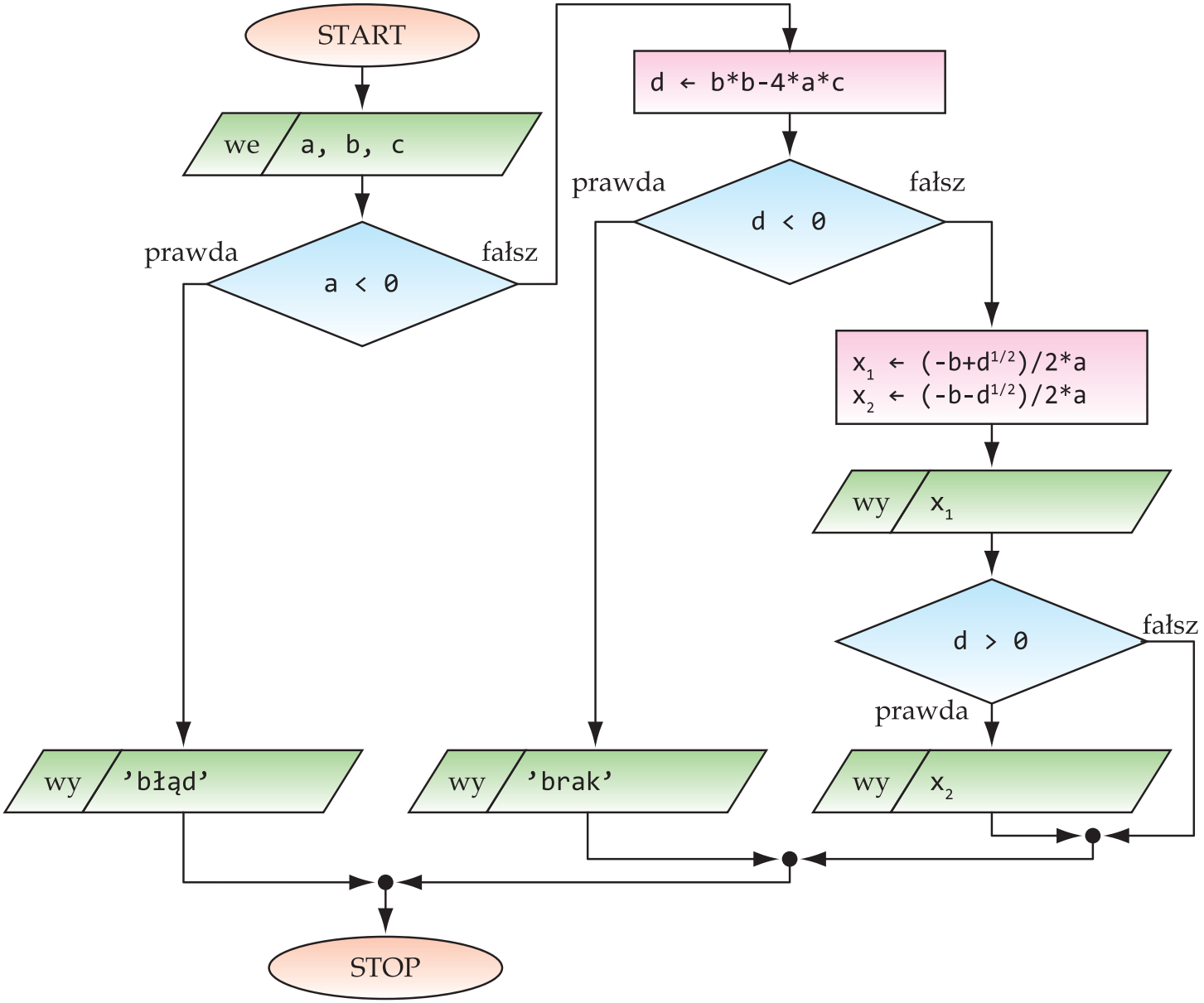
Algorytmy można podzielić na:

- algorytmy liniowe, jeśli wykorzystują operacje bezpośredniego następstwa,
- algorytmy warunkowe, jeśli wykorzystują operacje warunkowe,
- algorytmy iteracyjne, jeśli wykorzystują pętle.

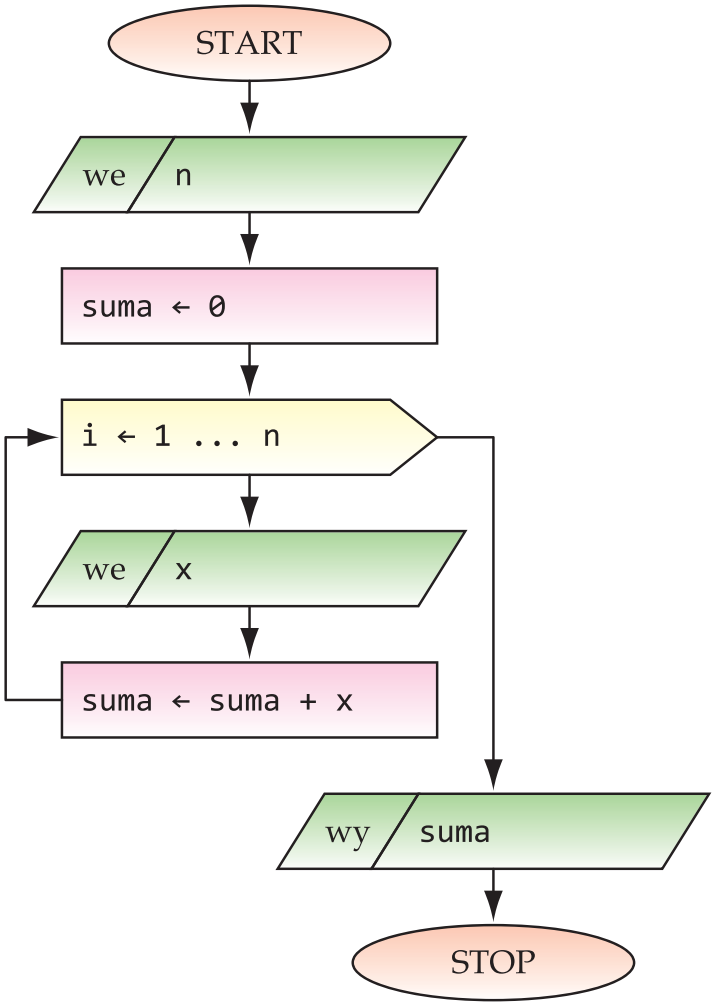
Algorytm szukania wartości minimalnej z trzech



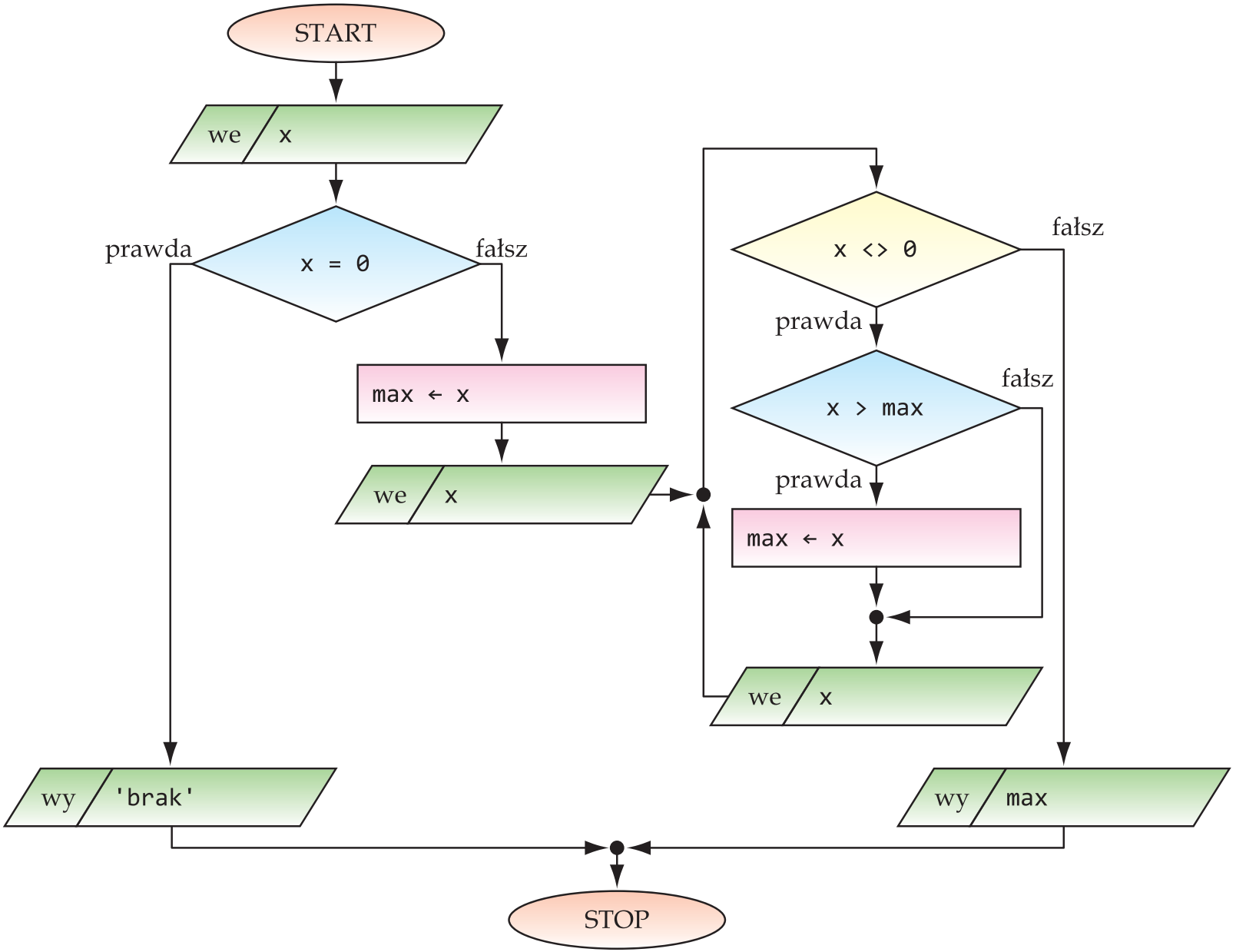
Algorytm obliczania pierwiastków równania kwadratowego



Algorytm sumowania



Algorytm szukania wartości ekstremalnej

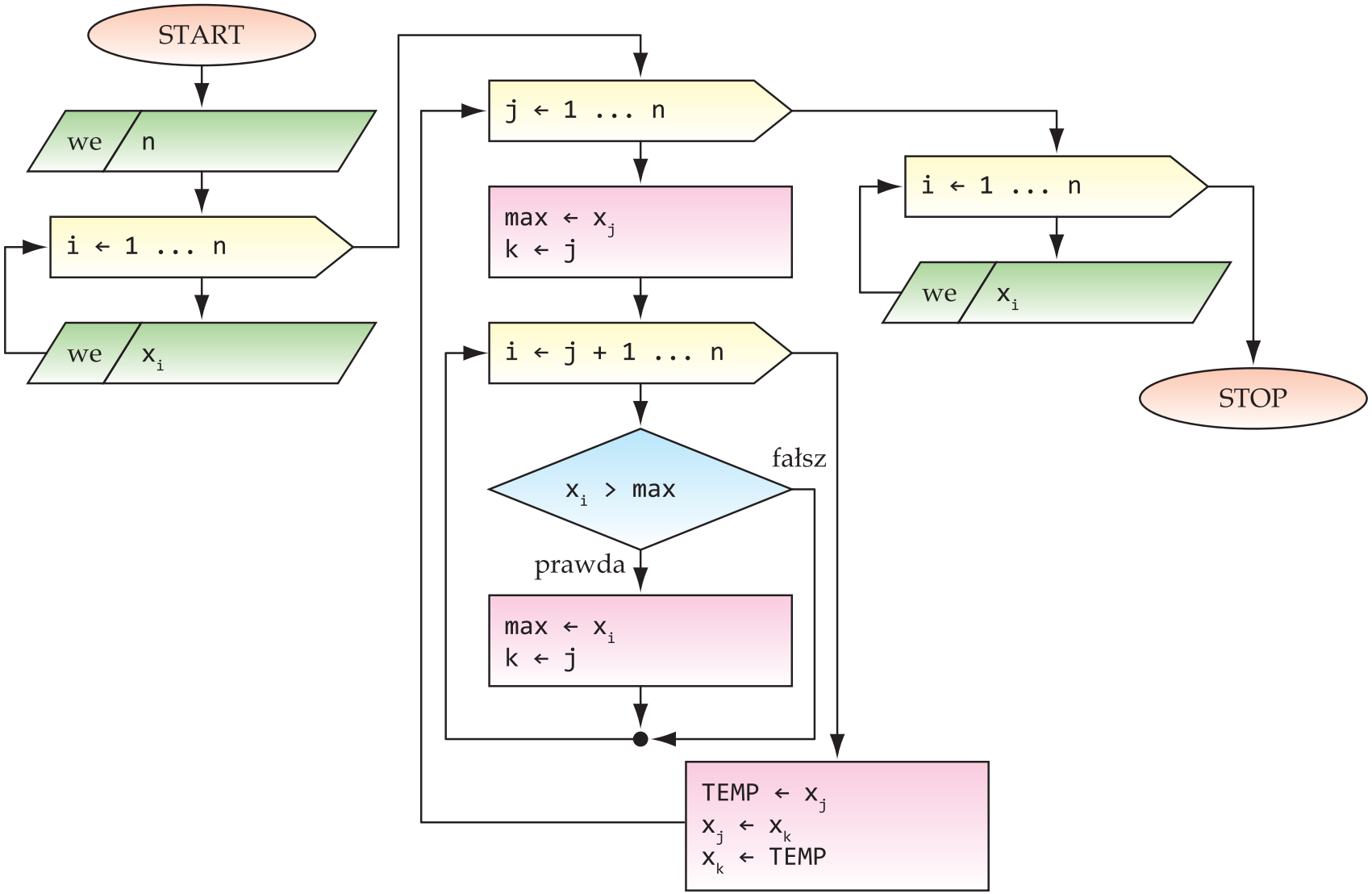


Algorytm sortowania przez wybór

Jest to chyba najbardziej intuicyjny algorytm sortowania. Polega on na wielokrotnym wyborze minimalnego elementu z coraz krótszego podciągu danych.

Przebieg:

- wybieranie minimum z ciągu elementów na pozycjach od 1 do n i zamienianie go z pierwszym elementem,
- wybieranie minimum z ciągu elementów na pozycjach od 2 do n i zamienianie go z drugim elementem (po tym kroku elementy na pozycjach od 1 do 2 są uporządkowane),
- wybieranie minimum z ciągu elementów na pozycjach $n-1$ i n i zamienianie z elementem na pozycji $n-1$ (po tej operacji elementy na pozycjach od 1 do $n-1$ są uporządkowane, a element na pozycji n jest maksymalny, czyli ciąg elementów na pozycjach od 1 do n jest uporządkowany).


START
we n
i ← 1 ... n
we x_i
j ← 1 ... n
max ← x_j
k ← j
i ← j + 1 ... n
x_i > max
fałsz
prawda
max ← x_i
k ← j
TEMP ← x_j
x_j ← x_k
x_k ← TEMP
i ← 1 ... n
we x_i
STOP

Podsumowanie

Budując algorytmy należy przestrzegać następujących zasad:

- algorytmy trzeba rozłożyć na jak najprostrze operacje podstawowe,
- każdy krok algorytmu dokładnie określić i przemyśleć,
- rozważyć wszystkie możliwe przypadki funkcjonowania algorytmu w zależności od danych wejściowych oraz operacji wewnętrznych,
- każdy algorytm powinien prowadzić do rozwiązania generując jedną lub wiele odpowiedzi na zadany problem.

Algorytm zdefiniowany z wykorzystaniem powyższych zasad powinien mieć jasno określony punkt startowy i punkt końcowy.

Dla dowolnego systemu liczenia istnieje zbiór cyfr, z których tworzone są liczby.

Systemy liczbowe dzieli się na:

- pozycyjne,
- niepozycyjne.

Systemy niepozycyjne

W systemach **niepozycyjnych** poszczególne cyfry zachowują swą wartość liczbową bez względu na miejsce jakie zajmują w liczbie (np. system rzymski – siedem znaków: I, V, X, L, C, D, M).

XII = 10 (X) + 1 (I) + 1 (I) = 12
IX = 10 (X) - 1 (I) = 9
MCDX = 1000 (M) + 500 (D) - 100 (C) + 10 (X) = 1410

Systemy pozycyjne

W systemach **pozycyjnych** wartość liczbowa cyfry zależy od umiejscowienia (pozycji) w liczbie (np. system dziesiętny, dwójkowy).

Liczba różnych cyfr systemu nazywa się jego **podstawą** P.

Wartość liczbowa cyfry określona jest przez **wagę**. Waga na pozycji n równa jest podstawie P podniesionej do potęgi n.

Sposób zapisu liczby L w dowolnym systemie pozycyjnym jest bardzo prosty:

$$L = a_{n-1} \cdot P^{n-1} + a_{n-2} \cdot P^{n-2} + \ldots + a_0 \cdot P^0 = \sum_{i=0}^{n-1} a_i \cdot P^i.$$

Dla systemu **dziesiętnego** (decymalnego):

$$P = 10, \quad a_n = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\},$$

$$142 = 1 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0 = 1 \cdot 100 + 4 \cdot 10 + 2 \cdot 1.$$

Dla systemu **dwójkowego** (binarnego):

$$P = 2, \quad a_n = \{0, 1\},$$

$$10010_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 18.$$

Dla systemu **szesnastkowego** (heksadecymalnego):

$$P = 16, \quad a_n = \{0, 1, 3, 4, 5, 6, 7, 8, 9, \mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}, \mathrm{f}\},$$

$$3e8_{16} = 3 \cdot 16^2 + 14 \cdot 16^1 + 8 \cdot 16^0 = 3 \cdot 256 + 14 \cdot 16 + 8 \cdot 1 = 1000.$$

Konwersja liczb naturalnych

$$190 = 1 \cdot 10^2 + 9 \cdot 10^1 + 0 \cdot 10^0,$$

$$190 = ?_2$$

dzielenie, wynik		
190 / 2	= 95	reszta = 0
95 / 2	= 47	reszta = 1
47 / 2	= 23	reszta = 1
23 / 2	= 11	reszta = 1
11 / 2	= 5	reszta = 1
5 / 2	= 2	reszta = 1
2 / 2	= 1	reszta = 0
1 / 2	= 0	reszta = 1

$$190 = 10111110_2.$$

$$10111110_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 =$$

$$= 1 \cdot 128 + 0 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1.$$

Używając jednego bajtu, można zapisać liczę z zakresu $0 \ldots 255$, czyli

$$00000000_2 \ldots 11111111_2$$

$$00_{16} \ldots ff_{16}.$$

Działania arytmetyczne na liczbach w systemie dwójkowym i szesnastkowym są odpowiednikiem działań w systemie dziesiętnym i opierają się na elementarnych działaniach, np.:

- $1_2 + 0_2 = 1_2$,
- $1_2 + 1_2 = 10_2$,
- $1_2 \cdot 0_2 = 0_2$,
- $1_2 \cdot 1_2 = 1_2$,
- $10_2 - 1_2 = 1_2$.

Przykłady:

```
    1 1 1 1 1 1
    1 1 1 1 1 1 1
+       1 0 0 1 1
-------------------
  1 0 0 1 0 0 1 0
```

```
    1 1 1 1 1 1 1
-       1 0 0 1 1
-------------------
    1 1 0 1 1 0 0
```

```
          1 1 0 1
×         1 0 1 1
-------------------
          1 1 0 1
        1 1 0 1
      0 0 0 0
+   1 1 0 1
-------------------
1 0 0 0 1 1 1 1
```

```
      1 1 0
-------------
  1 1 0 1 / 1 0
- 1 0
-------
  0 1 0 1
-   1 0
-------
  0 0 0 1
-     1 0
-------
  0 0 0 1
```

Jednostki informacji

Najmniejszą jednostką informacji potrzebna do określenia, który z dwóch równie prawdopodobnych stanów przyjął układ jest **bit**, **b** (*BInary digiT*). Bit odpowiada informacji Tak – Nie, Prawda – Fałsz, 1 – 0.

Jest to również najmniejsza jednostka informacji używana w odniesieniu do sprzętu komputerowego. Bit przyjmuje wartości wtedy **0** lub **1**.

Wyższe jednostki to informacji:

- półbajt (*nybble*) – 4 bity,
- **oktet** (*octet*) – 8 bitów,
- **bajt** (*byte*) **B** – pierwotnie liczba bitów przetwarzana jednocześnie przez komputer lub adresowana przez procesor, obecnie używany wyłącznie do oznaczania 8 bitów (czyli oktetu).

Często używa się też:

- **słowo** (*word*) to zwykle 16 bitów (2 bajty) – jednostka informacji, na której operuje komputer, mogą zdarzać się słowa o innej długości, np.: 4, 8, 16 bajtów,
- **słowo procesora** – jednostka informacji o długości naturalnej dla procesora,
- **słowo pamięci** – jednostka informacji możliwa do przesłania w jednym cyklu transmisji do/z pamięci, zwykle 64 b, czasem 128 b.

Do określenia większej liczby bajtów stosuje się często (**niepoprawnie**) przedrostki **kilo**, **mega**, **giga** itd. Są to przedrostki dziesiętne układu SI, będące wielokrotnościami liczby 10 ($10^{3 \cdot n}$) a nie liczby 2:

- **kilobajt** (*kilobyte*), **kB** – $10^3 = 1000^1$ – 1 tysiąc bajtów,
- **megabajt** (*megabyte*), **MB** – $10^6 = 1000^2$ – 1 milion bajtów,
- **gigabajt** (*gigabyte*), **GB** – $10^9 = 1000^3$ – 1 miliard bajtów,
- **terabajt** (*terabyte*), **TB** – $10^{12} = 1000^4$ – 1 bilion bajtów.

W celu odróżnienia przedrostków o mnożniku 1000 od przedrostków o mnożniku 1024, pojawiła się propozycja ujednoznacznienia, opracowana przez IEC, polegająca na dodawaniu litery **i** po symbolu przedrostka dwójkowego oraz **bi** po jego nazwie.

Nowe przedrostki nazywane zostały przedrostkami **dwójkowymi** (binarnymi). Jednak ta propozycja rozwiązania problemu niejednoznaczności przedrostków nie została przyjęta przez wszystkie środowiska. Przedrostki dwójkowe są wielokrotnościami liczby 2 ($2^{10 \cdot n}$):

- **kibibajt** (*kibibyte*), **KiB** – $2^{10} = 1024^1$ – 1 tysiąc 24 bajty,
- **mebibajt** (*mebibyte*), **MiB** – $2^{20} = 1024^2$ – 1 milion 48 tysięcy 576 bajtów,
- **gibibajt** (*gibibyte*), **GiB** – $2^{30} = 1024^3$ – 1 miliard 73 miliony 741 tysięcy 824 bajtów,
- **tebibajt** (*tebibyte*), **TiB** – $2^{40} = 1024^4$ – 1 bilion 99 miliardów 511 milionów 627 tysięcy 776 bajtów.

Współczesne komputery są używane do przetwarzania danych różnych typów – liczbowych, logicznych, tekstowych, a także obrazów i dźwięków. Działają w oparciu o system binarny.

- Wartości logiczne (prawda/fałsz).
- Znaki pisarskie.
- Liczby
  - całkowite
    - nieujemne
    - ze znakiem
  - niecałkowite
    - stałopozycyjne
    - zmiennopozycyjne
- Dźwięki, sygnały jednowymiarowe.
- Obrazy rastrowe.

Wszystkie dane, na których operuje komputer, są zapisane w postaci tbciągów cyfr binarnych – bitów, interpretowanych najczęściej jako liczby binarne.

Wszelkie dane o charakterze innym niż liczby, muszą być zapisane (zakodowane) w postaci liczb lub grup liczb.

Komputer przetwarza wyłącznie grupy bitów, tworząc liczby binarne, których długość wynosi $8 \cdot 2^n$ bitów (np. 8, 16, 32, 64).

Dane alfanumeryczne

Dane alfanumeryczne, tekstowe – mają postać znaków pisarskich – liter, cyfr, znaków przestankowych i innych symboli. W komputerze są one reprezentowane przez liczby, określające pozycję danego symbolu w tablicy kodowej.

We współczesnych komputerach używa się kilku standardów kodowania znaków pisarskich, najpopularniejsze to:

- ASCII,
- UNICODE.

ASCII

ASCII – (*American Standard Code for Information Interchange*), 7-bitowy kod przyporządkowujący liczby z zakresu $0 \ldots 127$ literom (alfabetu angielskiego), cyfrom, znakom przestankowym i innym symbolom oraz poleceniom sterującym. Został opracowany dla urządzeń dalekopisowych.

Na przykład litera **a** jest kodowana liczbą 97, a znak spacji jest kodowany liczbą 32.

Litery, cyfry oraz inne znaki drukowane tworzą zbiór znaków ASCII. Jest to 95 znaków o kodach 32 ... 126. Pozostałe 33 kody (0 ... 31 i 127) to tzw. kody sterujące służące do sterowania urządzeniem odbierającym komunikat, np. drukarką lub terminalem.

- Kody sterujące zajmują pozycje: 0 ... 31, w tym:
  - **CR** - powrót na początek wiersza: kod `13`,
  - **LF** – przejście do następnego wiersza: kod `10`,
  - inne ważne: **HT** (tabulacja pozioma), **FF** (rozpoczęcie nowej strony), **BEL** (sygnał dźwiękowy), **VT** (tabulacja pionowa), **BSP** (cofnięcie o jeden znak).
- **Spacja**: kod `32` (`0x20`).
- **Cyfry** $0 \ldots 9$: kody `48` ... `57` (`0x30` ... `0x39`).
- **Litery** w kolejności alfabetycznej:
  - wielkie: kody `65` ... `90` (`0x41` ... `0x5a`),
  - małe: kody `97` ... `122` (`0x61` ... `0x7a`).
- **Kasowanie znaku**: kod `127` (`0x7f`).

Ponieważ kod ASCII jest 7-bitowy, a większość komputerów operuje na 8-bitowych bajtach, dodatkowy bit został wykorzystany na powiększenie zbioru kodowanych znaków do 256 symboli.

Powstało wiele różnych rozszerzeń ASCII wykorzystujących ósmy bit. W kodach tych pierwsze 128 pozycji jest identyczne, jak w kodzie ASCII, a następne 128 pozycji zawiera znaki dodatkowe, np. litery akcentowane, rozszerzony zestaw symboli matematycznych, litery alfabetów narodowych (słowiańskie, skandynawskie, cyrylica, etc.).

Istnieje wiele rozszerzeń tej rodziny, używanych w różnych częściach świata. W Polsce najpowszechniej używa się kodów **ISO8859-2** oraz Microsoft **CP1250**, nazywanych stronami kodowymi.

BIN, DEC, HEX, znak				BIN, DEC, HEX, znak			
00000000	0	00	NUL (*Null*)	01000001	65	41	A
00000001	1	01	SOH (*Start Of Heading*)	01000010	66	42	B
00000010	2	02	STX (*Start of Text*)	01000011	67	43	C
00000011	3	03	ETX (*End of Text*)	01000100	68	44	D
...				...			
00110000	48	30	0	01110111	119	77	w
00110001	49	31	1	01111000	120	78	x
00110010	50	32	2	01111001	121	79	y
00110011	51	33	3	011 1010	122	7A	z
...				...			
00111101	61	3D	=	01111100	124	7C	|
00111110	62	3E	>	01111101	125	7D	}
00111111	63	3F	?	01111110	126	7E	~
01000000	64	40	@	01111111	127	7F	DEL (*Delete*)
...				...			

Unicode

Unicode – uniwersalny komputerowy zestaw znaków mający w zamierzeniu obejmować wszystkie znakówi pisarskie zapisu fonetycznego (głoskowego) używanych na całym świecie. Liczba pozycji kodowych jest praktycznie nieograniczona, obecnie jest zdefiniowanych kilkadziesiąt tysięcy znaków.

Definiują go dwa standardy – Unicode oraz **ISO10646**. Znaki obu standardów są identyczne. Standardy te różnią się w drobnych kwestiach, m.in. Unicode określa sposób składu.

Unicode jest rozwijany jest przez konsorcjum, w którego skład wchodzą firmy komputerowe, producenci oprogramowania, instytuty naukowe, agencje międzynarodowe oraz grupy zainteresowanych użytkowników. Konsorcjum współpracuje z organizacją ISO.

Standard Unicode obejmuje przydział przestrzeni numeracyjnej poszczególnym grupom znaków, nie obejmuje zaś sposobów bajtowego kodowania znaków.

Jest kilka metod kodowania, oznaczanych skrótowcami **UCS** (*Universal Character Set*) i **UTF** (*Unicode Transformation Format*). Do najważniejszych należą:

- UTF-32/UCS-4,
- UTF-16,
- UTF-8.

Kody pierwszych 256 znaków Unicode pokrywają się z kodami ISO Latin 1 (czyli ISO8859-1), przez co kody pierwszych 128 znaków pokrywają się z kodami ASCII.

Należy jednak pamiętać, że jest to zbieżność wyłącznie numerów przyporządkowanych konkretnym znakom, wartości bajtów użytych do ich zapisania mogą się różnić od tych, które uzyska się stosując Latin 1 lub ASCII.

UTF-8 – zmiennej długości system kodowania znaków dla Unicode. Może reprezentować każdy znak w systemie Unicode. Kodowanie zostało zaprojektowane dla zapewnienia zgodności z ASCII.

Zalety kodowania UTF-8:

- każdy tekst w ASCII jest tekstem w UTF-8,
- żaden znak spoza ASCII nie zawiera bajtu z ASCII,
- typowy tekst ISO Latin-*x* rozrasta się w bardzo niewielkim stopniu po przekonwertowaniu do UTF-8,
- znaki o kodzie różnym od 0 nie zawierają bajtu 0, co pozwala stosować UTF-8 w ciągach zakończonych zerem,
- o każdym bajcie wiadomo, czy jest początkiem znaku, czy też leży w jego środku,
- nie zawiera bajtów 0xff i 0xfe, więc łatwo można go odróżnić od tekstu UTF-16.
- brak problemów z uporządkowaniem bajtów (*endianness*),
- jest domyślnym kodowaniem w ML (również w jego aplikacjach: HTML, SVG, XSL, CML, MathML).

Dźwięk i obraz

Dźwięk jest zapamiętywany w postaci wartości napięcia reprezentującego chwilowe ciśnienie akustyczne. Wartość ta jest kwantowana z użyciem rozpiętości zwykle od 8 bitów do 32 bitów z częstotliwością zależną od potrzeb, zwykle od 8 kHz do 48 kHz (próbkowanie).

Obraz rastrowy jest zapisany w postaci prostokątnej macierzy punktów (**pikseli**). Każdemu pikselowi odpowiada jeden kolor, zapisany w postaci składowych – jasności świateł (barw) podstawowych. Wartości jasności zapisane są w postaci liczb.

Dane logiczne

Najprostszy typ danych stanowią dane **logiczne**. Mogą one przyjmować dwie wartości. Bajtowe adresowanie danych używane w komputerach oraz fakt, że wiele komputerów traktuje jako podstawowy format danych słowo 32-bitowe powodują, że dane logiczne są zwykle zapisywane w postaci bajtów lub słów, pomimo, że do ich zapisu wystarczyłby pojedynczy bit.

Należy zwrócić uwagę na reprezentację wartości *prawda* i *fałsz* w różnych językach programowania. Wartość *fałsz* jest zwykle reprezentowana przez słowo o wszystkich bitach równych 0.

Wartość *prawda* może być reprezentowana przez liczbę całkowitą równą 1, liczbę całkowitą o dowolnej wartości różnej od zera (również ujemną), słowo o wszystkich bitach równych 1.

Liczby całkowite nieujemne

W dalszej będziemy posługiwali się założeniem, że dane reprezentowane są przez słowo komputera, w którym poszczególne bity zostały ponumerowane od prawej do lewej strony.

NBC

Kod **NBC** (*Natural Binary Code*, NKB, naturalny kod binarny) jest w zasadzie tym samym co pozycyjny system dwójkowy. Numer bitu jest równy wykładnikowi jego wagi binarnej.

$$L_{NCB} = \sum_{i=0}^{n-1} b_i \cdot 2^i.$$

Wartości wag w kodzie NBC (dla bajtu)

waga	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
cyfra	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

Liczby całkowite ze znakiem

2C

Zapis **2C** (*Two's Complement*, U2, uzupełnień do 2) jest najczęściej stosowanym zapisem liczb całkowitych. Jest on podobny do NKB, z tą różnicą, że najbardziej znaczący bit ma wagę ujemną. Typ `int` jest we współczesnych komputerach implementowany jako zapis 2C. Negacja liczby w tym systemie polega na negacji bitowej i inkrementacji.

$$L_{2C} = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i.$$

Wartości wag w kodzie 2C

waga	$-(2^8) = -128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
cyfra	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

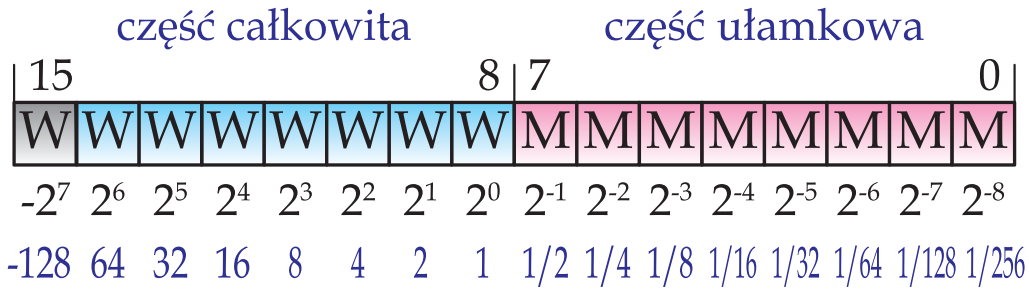
Liczby niecałkowite

System stałopozycyjny

Do zapisywania liczb ułamkowych i mieszanych można użyć zapisu **stałopozycyjnego**. W zapisie tym liczba jest reprezentowana przez słowo binarne, w którym pewne, z góry określone liczby bitów reprezentują część całkowitą i część ułamkową liczby.

Odpowiada to interpretacji zapisu całkowitoliczbowego, pomnożonej przez wartość będącą ujemną potęgą liczby 2.

Do zapisu liczb bez znaku używa się jako bazowej postaci NKB, a do zapisu liczb ze znakiem kodu U2 (zwykle).



Komputery zazwyczaj nie obsługują w szczególny sposób zapisów stałopozycyjnych. Podstawowe operacje są wykonywane tak samo, jak na liczbach całkowitych, odmienna jest jedynie interpretacja zapisu, za którą jest odpowiedzialny wyłącznie programista.

Zapis zmiennopozycyjny dla systemu dziesiętnego

Zapis **zmiennopozycyjny** umożliwia zapisywanie liczb całkowitych i ułamkowych o bardzo dużym zakresie dynamiki wartości bezwzględnych.

Każda liczba może być zapisana na kilka sposobów, różniących się położeniem przecinka oddzielającego część całkowitą od ułamkowej i wartością wykładnika:

$$-1,2345 \cdot 10^5 \qquad -0,12345 \cdot 10^6 \qquad -12,345 \cdot 10^4.$$

Wartość liczby zmiennoprzecinkowej jest obliczana według wzoru:

$$L = Z \cdot M \cdot B^W,$$

gdzie:

- Z (*sign*) – znak liczby, 1 lub -1,
- M (*mantissa*) – mantysa, liczba ułamkowa,
- B (*base*) – podstawa systemu liczbowego,
- W (*exponent*) – wykładnik (cecha), liczba całkowita.

Postacią **znormalizowaną** nazwiemy liczbę, która ma część całkowitą części znaczącej wyrażoną przez pojedynczą cyfrę różną od zera.

Aby zapisać (przechować) liczbę, musimy zapisać jej znak, część znaczącą oraz wykładnik, który jest liczbą całkowitą ze znakiem. Podstawa systemu jest ustalona i jej zapis jest zbędny.

$$-1,2345e5$$

W postaci znormalizowanej nie da się zapisać zera, ponieważ zero nie ma żadnej cyfry znaczącej różnej od 0.

Binarny zapis zmiennopozycyjny

Obecnie niemal wszystkie komputery posługują się binarnym zapisem zmiennopozycyjnym zgodnym ze standardem IEEE754.

Standard zakłada, że, o ile tylko jest to możliwe, liczby zapisuje się w postaci znormalizowanej. **Bazą** systemu jest liczba 2, **wykładnik** określa potęgę liczby 2.

Ponieważ w systemie binarnym jedyną cyfrą różną od zera jest jedynka, każda liczba w postaci znormalizowanej ma część całkowitą równą 1, nie ma więc potrzeby zapisywania jej, zapisuje się tylko część ułamkową.

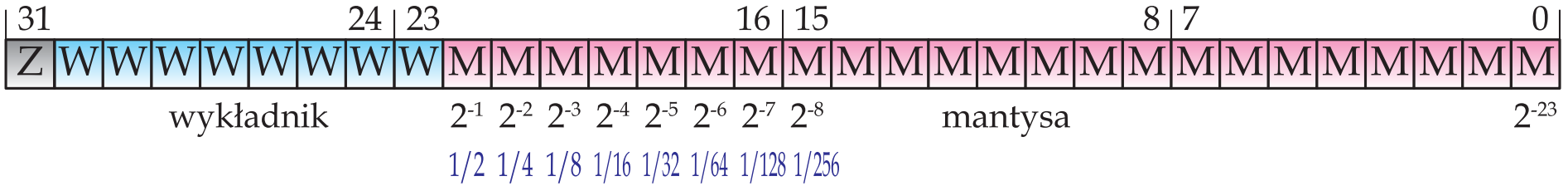
Wykładnik jest liczbą całkowitą ze znakiem. W IEEE754 wykładnik jest zapisywany w kodzie **spolaryzowanym** (biased), w którym wartość podkładu jest określona wzorcem bitowym `01` ... `11`, o liczbie bitów równej szerokości pola bitowego wykładnika. Dwie wartości pola wykładnika są zarezerwowane i oznaczają, że zapis nie reprezentuje postaci znormalizowanej.

Bity wykładnika o wzorcu `00` ... `00` oznaczają zapis zdenormalizowany. Wartość wykładnika jest w tym przypadku taka sama, jak przy zapisie znormalizowanym z wzorcem wykładnika `00` ... `01`, a część całkowita części znaczącej ma wartość 0 (a nie 1 jak w postaci znormalizowanej).

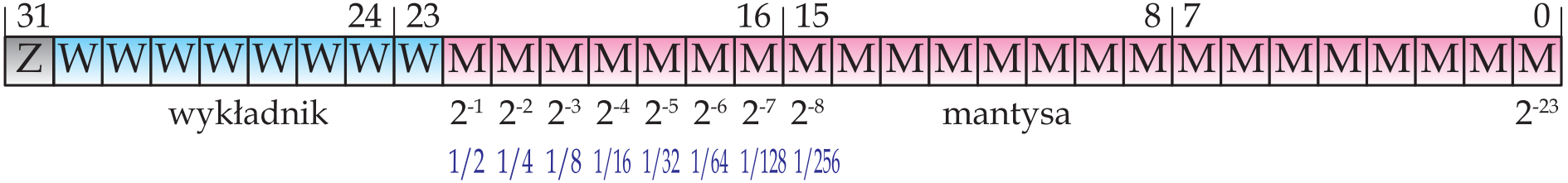
Pole wykładnika o wzorcu `11` ... `11` oznacza nie-liczby: wartości **nieskończone** i wartości **błędne**.

Liczba pojedynczej precyzji zajmuje 4 bajty (32 bity).

Mantysa ma 23 bity, wykładnik ma 8 bitów, znak 1 bit, a nadmiar wynosi 127.



Mantysa jest **znormalizowana**, tj. należy do przedziału $[1, 2)$ (przedział prawostronnie otwarty). Jeżeli M jest stałe, a W zmienia się, wówczas przesunięciu ulega przecinek – stąd właśnie pochodzi nazwa tej reprezentacji.



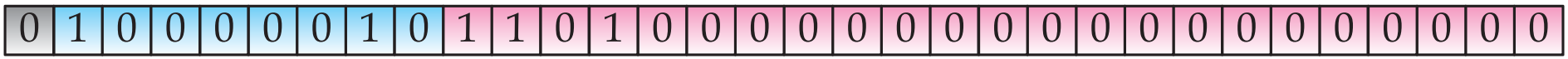
Procedura zapisu liczby L:

- określenie znaku, bit 31 = `1` jeżeli L jest ujemna, `0` jeśli dodatnia,
- znalezienie największej liczby postaci 2^W, mniejszej niż L,
- zapisanie wykładnika z nadmiarem $W + 127$,
- dzielenie liczby $L/2^W$ (wynik $1.xxx$),
- odjęcie 1 i szukanie mantysy,
- zaznaczenie bitu jako `1` jeśli po odjęciu $1/2$ (bit + 1) mamy wartość nieujemną, jeśli wartość jest ujemna, bit oznaczany jest jako `0`, i ignorowana operacja odejmowania,
- procedura jest powtarzana aż w wyniku odejmowania otrzymamy 0 lub dojdzie do bitu nr 0.

Przykład dla liczby $14,5$:

- liczba jest dodatnia $\Rightarrow$ bit 31 = `0`,
- największa liczba 2^W mniejsza niż 14,5 to $3^3 = 8 \Rightarrow W = 3$,
- zapisanie wykładnika $127 + W = 130 \Rightarrow$ `10000010`,
- dzielenie $14,5/2^3 = 1,8125$,
- odejmowanie $1,8125 - 1 = 0,8125$,
  - $0,8125 - 1/2 = 0.3125 \Rightarrow$ bit 23 = `1`,
  - $0,3125 - 1/4 = 0.0625 \Rightarrow$ bit 22 = `1`,
  - $0,0625 - 1/8 = -0.0625 \Rightarrow$ bit 21 = `0` (ignorowanie operacji),
  - $0,0625 - 1/16 = 0 \Rightarrow$ bit 20 = `1`,
- pozostałe bity mantysy = `0`.

14,5



Niektóre wartości specjalne

wartość,	wykładnik,	mantysa,	uwagi
NaN	11...11	xx...xx	symbol nie reprezentuje wartości liczbowej, powstaje zazwyczaj w wyniku niedozwolonej operacji (np. pierwiastkowanie liczby ujemnej)
INF	11...11	00...00	bit znaku decyduje o znaku wartości, rozróżnia się $+\infty$ i $-\infty$
0	00...00	xx...xx	wartości zdenormalizowane (*denormalized numbers*) lub częściej *subnormal numbers*, wykładnik jest równy zero, mantysa różna od 0, mantysa nie posiada domyślnej części całkowitej 1
0	00...00	00...00	bit znaku decyduje o znaku wartości, jest wynikiem operacji w przypadku wystąpienia nadmiaru (przepełnienia), przy dzieleniu przez 0, itp., może być dodatnia lub ujemna

Standard IEEE754, definiuje klasy liczb:

- pojedynczej precyzji (`single`),
- podwójnej precyzji (`double`),
- poczwórnej precyzji (`quad`, od 2008).

Podstawowym formatem jest format typu `double` – 64-bitowy (podwójnej precyzji).

Format 32-bitowy jest używany w zastosowaniach, gdzie wymagana precyzja jest niewielka, ma on tylko 23 bity znaczące (typ `single`).

Liczba **pojedynczej** precyzji: mantysa 23 bity, wykładnik 8 bitów, znak 1 bit.

Liczba **podwójnej** precyzji: mantysa 52 bity, wykładnik 11 bitów, znak 1 bit.

Liczba **poczwórnej** precyzji: mantysa 112 bitów, wykładnik 15 bitów, znak 1 bit.

Format 80-bitowy (`extended`) jest używany jednostkach zmiennopozycyjnych procesorów rodziny x86.

Precyzja – liczba miejsc po przecinku jest określona przez mantysę.

Pojedyncza precyzja
Mantysa ma 23 bity, $1/2^{23} \approx 1,2 \cdot 10^{-7} \Rightarrow$ 7 cyfr po przecinku.

Podwójna precyzja
Mantysa ma 52 bity, $1/2^{52} \approx 2,2 \cdot 10^{-16} \Rightarrow$ 16 cyfr po przecinku.

Rozszerzona precyzja
Mantysa ma 64 bity, $1/2^{64} \approx 5,4 \cdot 10^{-20} \Rightarrow$ 20 cyfr po przecinku.

Poczwórna precyzja
Mantysa ma 112 bitów, $1/2^{112} \approx 1,9 \cdot 10^{-34} \Rightarrow$ 34 cyfry po przecinku.

Operacje arytmetyczne

Mamy dwie liczby zmiennoprzecinkowe: $L_1 = M_1 \cdot B^{W_1}$ i $L_2 = M_2 \cdot B^{W_2}$, przy czym $L_1 > L_2$, wówczas:

- dodawanie

$$L_1 + L_2 = \left(M_1 + M_2 \cdot B^{W_2 - W_1}\right) \cdot B^{E_1},$$

- odejmowanie

$$L_1 - L_2 = \left(M_1 - M_2 \cdot B^{W_2 - W_1}\right) \cdot B^{E_1}.$$

Mamy dwie liczby zmiennoprzecinkowe: $L_1 = Z_1 \cdot M_1 \cdot B_1^W$ i $L_2 = Z_2 \cdot M_2 \cdot B_2^W$, wówczas:

- mnożenie

$$L_1 \cdot L_2 = (Z_1 \cdot Z_2) \cdot (M_1 \cdot M_2) \cdot B^{W_1 + W_2},$$

- dzielenie

$$L_1 / L_2 = (Z_1 \cdot Z_2) \cdot (M_1 / M_2) \cdot B^{W_1 - W_2}.$$

Jeśli liczby mają różne wykładniki, to podczas dodawania mantysa liczby o mniejszym wykładniku musi zostać **zdenormalizowana** – we wzorze jest to przemnożenie M_2 przez czynnik $B^{W_2-W_1}$.

W szczególnym przypadku, jeśli $W_1 - W_2$ jest większe niż liczba cyfr mantysy, to po denormalizacji mantysa będzie miała wartość 0, a liczba o mniejszym wykładniku nie wpłynie na wynik dodawania bądź odejmowania.

Liczby o skończonej reprezentacji dziesiętnej mogą mieć nieskończoną reprezentację binarną (np.: $0,1$ lub $0,3$).

Reprezentacja zmiennopozycyjna jest reprezentacją przybliżoną, a wyniki operacji są w rzeczywistości przybliżeniami. Oznacza to, że w praktyce nie można stosować relacji równości w odniesieniu do liczb zmiennopozycyjnych ($|a - b| < \epsilon$).

Arytmetyka liczb zmiennopozycyjnych **nie jest łączna**, dla x i y może zachodzić:

$$(x + y) + z \neq x + (y + z),$$
$$(x \cdot y) \cdot z \neq x \cdot (y \cdot z),$$

nie jest też rozdzielna, czyli:

$$x \cdot (y + z) \neq (x \cdot y) + (x \cdot z).$$

Innymi słowy, kolejność wykonywania operacji arytmetycznych ma znaczenie i wpływa na końcowy wynik.

Po co mi programowanie?

Linus Torvalds

Computer programming is not for everyone. I think it's reasonably specialized, and nobody really expects most people to have to do it.

Steve Jobs

Everybody in this country should learn how to program a computer, because it teaches you how to think.

Celem nauki programowania jest wykształcenie umiejętności **myślenia komputacyjnego** (*computational thinking*), które obejmuje myślenie algorytmiczne w rozwiązywaniu problemów oraz umiejętność programowania rozszerzone na wszystkie obszary działalności ludzkiej. Takie podejście przede wszystkim pozwala zwiększyć efektywność i ułatwić pracę.

Nawet życiowe problemy i decyzje można potraktować jako problem algorytmiczny.

Języki programowania lub **makropolecenia** udostępniają szereg narzędzi pozwalających na przyśpieszenie i zwiększenie efektywności pracy, zrobienie czegoś w łatwiejszy sposób, a nawet utworzenie kompletnie nowych narzędzi.

Rozwój języków programowania znacznie obniżył próg umiejętności jakie należy posiadać, żeby zacząć naukę. Nie trzeba posiadać żadnych informacji na temat budowy komputera, nie trzeba mieć solidnych podstaw matematycznych (choć te są przydatne), nie trzeba mieć superkomputera ani kupować dodatkowego drogiego oprogramowania.

Przypomnienie podstawowych terminów

Program komputerowy (aplikacja) – sekwencja symboli (zrozumiałych dla komputera rozkazów) przeznaczonych do przetworzenia zgodnie z pewnymi regułami, zwanymi **językiem programowania**.

Program w postaci języka *zrozumiałego* dla człowieka nazywany jest **kodem źródłowym**, podczas gdy program wyrażony w postaci zrozumiałej dla maszyny (to jest za pomocą ciągu liczb, a bardziej precyzyjnie zer i jedynek) nazywany jest kodem maszynowym bądź postacią binarną (wykonywalną).

Program jest zazwyczaj wykonywany przez komputer, bezpośrednio – jeśli wyrażony jest w języku zrozumiałym dla danej maszyny lub pośrednio – gdy jest interpretowany przez inny program (interpreter).

Programy komputerowe można zaklasyfikować według ich zastosowań. Wyróżnia się zatem aplikacje użytkowe, systemy operacyjne, gry, kompilatory i inne. Programy wbudowane wewnątrz urządzeń określa się jako **firmware**.

W najprostszym modelu wykonanie programu (zapisanego w postaci zrozumiałej dla maszyny) polega na umieszczeniu go w pamięci operacyjnej komputera i wskazaniu procesorowi adresu pierwszej instrukcji.

Po tych czynnościach procesor będzie wykonywał kolejne instrukcje programu, aż do jego zakończenia.

Program komputerowy będący w trakcie wykonania nazywany jest **procesem** lub **zadaniem**.

Tworzenie programu komputerowego można podzielić na dwa etapy:

- Po zrodzeniu się **pomysłu** powinien powstać **algorytm**. Algorytm wymusza stosowanie podziału programu na funkcje. zmienne, obiekty, na których program będzie operował, jak również wprowadzenie procedur, które opisują wykonywane operacje.
- Algorytm należy zapisać w języku programowania, stosując dostępne struktury danych i funkcje – tworzenie **kodu źródłowego**. W trakcie tworzenia programu kod jest poddawany **debugowaniu** – wyszukiwanie błędów.

Językiem programowania nazywamy zestaw zasad tekstowego lub graficznego opisu algorytmu za pomocą przyjętych elementów języka.

Podobnie jak języki naturalne, język programowania składa się ze zbiorów reguł syntaktycznych oraz semantyki, które opisują, jak należy budować poprawne wyrażenia oraz jak komputer ma je rozumieć

Język programowania pozwala na precyzyjny zapis algorytmów oraz innych zadań, jakie komputer ma wykonać.

Podział języków programowania

Kod maszynowy (język maszynowy) – język programowania, w którym zapis programu wymaga **instrukcji** bezpośrednio jako liczb, które są rozkazami i danymi bezpośrednio pobieranymi przez procesor wykonujący ten program.

Jest dopasowany do konkretnego typu procesora i przeznaczony do bezpośredniego wykonania przez procesor. Analiza kodu maszynowego jest praktycznie niemożliwa przez człowieka.

Języki niskopoziomowe – przedstawiają one instrukcje udostępniane przez system komputerowy w postaci prostych oznaczeń (o ograniczonej liczbie, zakodowane w procesorze). Do języków niskopoziomowych należą **assemblery**.

Języki wysokopoziomowe – składnia i słowa kluczowe mają maksymalnie ułatwić rozumienie kodu programu dla człowieka, tym samym zwiększając poziom abstrakcji i dystansując się od budowy sprzętu komputerowego.

Kod napisany w języku wysokiego poziomu nie jest bezpośrednio *zrozumiały* dla komputera – większość kodu stanowią tak naprawdę normalne słowa (najczęściej w języku angielskim).

Języki wysokopoziomowe dzielimy na dwie grupy:

- interpretowane,
- kompilowane.

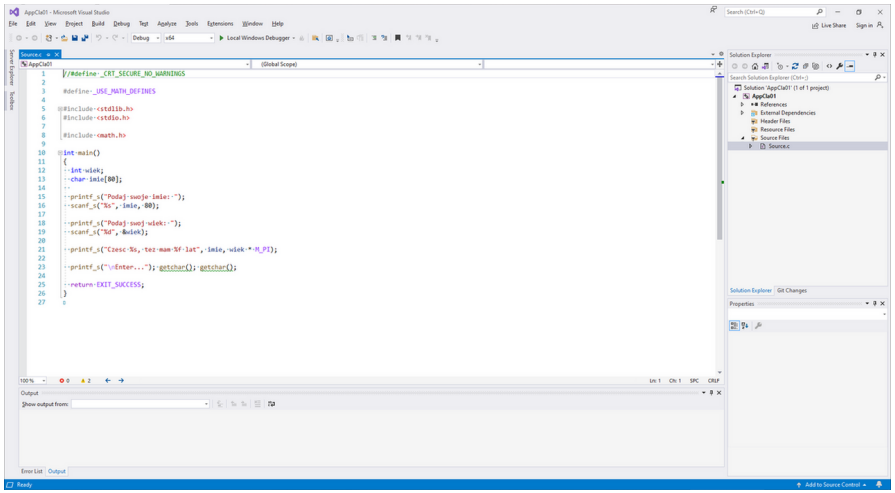
Języki interpretowane nie wymagają kompilacji tylko **interpretera**. Są przechowywane w postaci kodu źródłowego i dopiero podczas uruchomienia wczytywane, analizowane i wykonywane przez interpreter języka – **PHP**, **JavaScript**, **Python**, **PERL**. Programy przeznaczone do interpretacji często nazywane są **skryptami**.

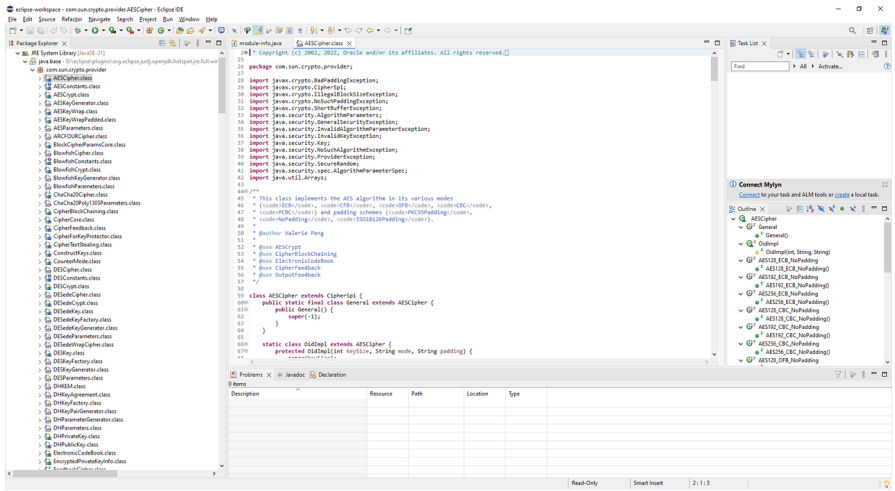
Języki kompilowane wymagają procesu kompilacji kodu źródłowego do postaci kodu maszynowego (postaci binarnej). Robi to specjalny program zwany **kompilatorem**, dzięki czemu możliwe staje się jego późniejsze uruchomienie. Języki kompilowane: **Pascal**, **C**, **C++**, **Fortran**, **Java**.

Do utworzenia programu w danym języku niezbędne są **edytor** tekstu, **debugger** i **kompilator**. Programy te mogą tworzyć integralne środowisko pracy, udostępniające kombinacje tych funkcji – mówimy wówczas o **środowisku programistycznym**.

Aby przyśpieszyć tworzenie aplikacji, szczególnie z interfejsem graficznym, tworzy się narzędzia **RAD** (*Rapid Application Development*), które umożliwiają tworzenie programów z gotowych komponentów (w sensie elementów interfejsu i funkcji z implementacją typowych algorytmów) na przykład:

- Microsoft VisualStudio,
- Eclipse IDE.





Syntaktyka i semantyka

Aby ciąg znaków mógł być rozpoznany jako program napisany w danym języku, musi spełniać reguły **syntaktyki** (składni). Składnia opisuje:

- rodzaje dostępnych symboli,
- zasady, według których symbole mogą być łączone w większe struktury.

Należy zauważyć, że na etapie przetwarzania składni w ogóle nie jest brane pod uwagę znaczenie poszczególnych symboli. W praktyce kod poprawny składniowo nie musi być poprawny semantycznie. Występuje tu analogia do języków naturalnych. Zdanie „Bźdźiągwy się mucioszą!” jest poprawne pod względem gramatycznym, lecz nie posiada żadnego znaczenia, ponieważ zostały w nim użyte nieistniejące słowa.

Semantyka języka programowania definiuje precyzyjnie znaczenie poszczególnych symboli oraz ich funkcję w programie. Semantykę najczęściej definiuje się słownie, ponieważ większość z jej zagadnień jest trudna lub wręcz niemożliwa do ujęcia w jakikolwiek formalizm.

Część błędów semantycznych można wychwycić już w momencie wstępnego przetwarzania kodu programu, np. próbę odwołania się do nieistniejącej funkcji, lecz inne mogą ujawnić się dopiero w trakcie wykonywania.

Błędy

W trakcie pisania kodu nie da się uniknąć błędów. Błędy mogą wynikać z niepoprawnego wpisania instrukcji, pominięcia kropek, przecinków, nawiasów, itp. Takie błędy noszą nazwę **syntaktycznych** (składniowych).

Oprócz błędów syntaktycznych, można spotkać jeszcze błędy **semantyczne** (znaczeniowe, wykonania) i **logiczne**.

Błędy wykonania występują w chwili odtwarzania procedur (programu) i mogą wynikać z próby uruchomienia nieistniejącej procedury, otwarcia nieistniejącego pliku lub złego typowania zmiennych.

Błędy logiczne zwykle nie wywołują żadnych komunikatów błędu. Chociaż program jest poprawny pod względem syntaktycznym i semantycznym, nie powoduje żadnych problemów kompilacji i uruchamiania to sam rezultat działania może być błędny.

Błędy logiczne powodują otrzymanie wyników innych niż się spodziewano. Są to zwykle błędy bardzo trudne do odnalezienia.

Języki programowania

Dla początkujących, problemem jest mnogość dostępnych obecnie języków programowania. Najbardziej klasyczne języki programowania to **C** i **C++**, popularne są **Java** i **C#**, modny jest **Python**, **JavaScript** i **Ruby**. Duża liczba użytkowników ceni **Go** i **Rust**.

Analitycy często używają jżyków pozwalających na szybkie pisanie aplikacji np. **Visual Basic** i eksploracji baz danych: **SQL**, czy dedykowane do analizy danych **R** lub do obliczeń numerycznych (analizy danych też) **MATLAB**, **Scilab**.

- `https://www.tiobe.com/tiobe-index/` – Wskaźnik popularności języków programowania.
- `https://insights.stackoverflow.com/survey/` – Ankiety serwisu społecznościowego programistów.

Paradygmat programowania

Paradygmat programowania – wzorzec programowania komputerów, który definiuje sposób patrzenia programisty na przepływ sterowania i wykonywanie programu komputerowego.

Przykładowo, w programowaniu **obiektowym** jest on traktowany jako zbiór współpracujących ze sobą obiektów, podczas gdy w programowaniu **funkcyjnym** definiujemy, co trzeba wykonać, a nie w jaki sposób.

Różne języki programowania mogą wspierać różne paradygmaty programowania. Przykładowo, **Smalltalk** i **Java** są ściśle zaprojektowane dla potrzeb programowania obiektowego, natomiast **Haskell** jest językiem funkcyjnym. Istnieją także języki wspierające kilka paradygmatów, np. **Python**.

Wiele paradygmatów jest dobrze znanych z tego, jakie praktyki są w nich zakazane, a jakie dozwolone.

Na przykład, ścisłe programowanie funkcyjne nie pozwala na tworzenie skutków ubocznych (dowolny efekt wyrażenia, lub wywołania funkcji, który wykracza poza zwrócenie wartości). W programowaniu strukturalnym nie korzysta się z instrukcji skoku.

Zależności między paradygmatami programowania mogą przybierać skomplikowane formy, ponieważ jeden język może wspierać wiele różnych paradygmatów. Na przykład, **C++** posiada elementy programowania proceduralnego, obiektowego oraz uogólnionego, co stanowi o nim, że jest hybrydowym językiem. To projektanci i programiści decydują, jak zbudować z nich w pełni działający program.

Przykłady paradygmatów programowania:

- programowanie **imperatywne**,
- programowanie **deklaratywne**,
- programowanie **proceduralne**,
- programowanie **strukturalne**,
- programowanie funkcyjne,
- programowanie **obiektowe**,
- programowanie uogólnione,
- programowanie **sterowane zdarzeniami**,
- programowanie logiczne,
- programowanie aspektowe,
- programowanie agentowe,
- programowanie modularne.

Programowanie imperatywne – paradygmat programowania, który opisuje proces wykonywania jako sekwencję instrukcji zmieniających stan programu, podobnie jak tryb rozkazujący, wyraża żądania jakichś czynności do wykonania.

Programy imperatywne składają się z ciągu komend do wykonania przez komputer. Rozszerzeniem (w sensie wbudowanych funkcji) i rodzajem (w sensie paradygmatu) programowania imperatywnego jest programowanie proceduralne.

Programowanie deklaratywne – rodzina paradygmatów programowania, które nie są z natury imperatywne. W przeciwieństwie do programów napisanych imperatywnie, programista opisuje warunki, jakie musi spełniać końcowe rozwiązanie (co chcemy osiągnąć), a nie szczegółową sekwencję kroków, które do niego prowadzą (jak to zrobić).

Przykłady języków: **XSLT**, **Prolog**.

Programowanie proceduralne to paradygmat programowania zalecający dzielenie kodu na procedury, czyli fragmenty wykonujące ściśle określone operacje.

Procedury nie powinny korzystać ze zmiennych globalnych (w miarę możliwości), lecz pobierać i przekazywać wszystkie dane (czy też wskaźniki do nich) jako parametry wywołania.

Programowanie strukturalne to paradygmat programowania zalecający hierarchiczne dzielenie kodu na bloki, z jednym punktem wejścia i jednym lub wieloma punktami wyjścia.

Chodzi przede wszystkim o nieużywanie instrukcji skoku. Dobrymi strukturami są np. instrukcje: warunkowe, pętle, wyboru.

Strukturalność zakłócają instrukcje typu: `break`, `continue`, `switch`, które jednak w niektórych przypadkach znacząco podnoszą czytelność kodu.

Praktycznie w każdym języku można programować strukturalnie, jednakże w niektórych jest to styl naturalny (np. **Pascal**).

Programowanie obiektowe (*Object-Oriented Programming*) – paradygmat programowania, w którym programy definiuje się za pomocą obiektów – elementów łączących stan (czyli dane, nazywane polami lub właściwościami) i zachowanie (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

Podejście to różni się od tradycyjnego programowania proceduralnego, gdzie dane i procedury nie są ze sobą bezpośrednio związane. Programowanie obiektowe ma ułatwić pisanie, konserwację i wielokrotne użycie programów lub ich fragmentów.

Największym atutem programowania, projektowania oraz analizy obiektowej jest zgodność takiego podejścia z rzeczywistością – mózg ludzki jest w naturalny sposób najlepiej przystosowany do takiego podejścia przy przetwarzaniu informacji. Przykłady języków: **C++**, **JAVA**.

Programowanie sterowane zdarzeniami – metodologia tworzenia programów komputerowych, która określa sposób ich pisania z punktu widzenia procesu przekazywania sterowania między poszczególnymi modułami tej samej aplikacji.

Programowanie sterowane zdarzeniami jest mocno powiązane ze środowiskami wieloprocesowymi, z graficznymi środowiskami systemów operacyjnych oraz z programowaniem obiektowym.

Paradygmat zakłada, że program jest cały czas bombardowany zdarzeniami, na które musi odpowiedzieć, i że przepływ sterowania w programie jest całkowicie niemożliwy do przewidzenia z góry.

Programowanie zdarzeniowe jest dominującym typem programowania związanego z graficznym interfejsem użytkownika (*Graphical User Interface*) – zdarzenia to naciśnięcia myszy, klawiszy, żądania odświeżenia przez system okienkowy, różne zdarzenia sieciowe i inne.

W programowaniu zdarzeniowym ważne jest żeby nie obsługiwać zbyt długo danego zdarzenia, bo blokuje się w ten sposób obsługę innych. W przypadku serwerów obniżyło by to znacznie wydajność, w przypadku GUI program zbyt wolno odpowiadałby na akcje użytkownika.

Można to osiągnąć za pomocą asynchronicznego I/O, wielowątkowości, rozbijania zdarzenia na podzdarzenia i wielu innych mechanizmów.

Python

Twórcą Pythona jest holender Guido van Rossum a sama nazwa pochodzi od popularnego serialu BBC **Latający Cyrk Monty Pythona**. Prace nad pierwszym interpreterem Pythona rozpoczęły się w 1989 roku jako następca języka ABC.

Od wersji 2.1 Python jest udostępniany jako projekt Open Source przez niedochodową organizację **Python Software Foundation** (PSF).

Guido van Rossum

Computer programming for everybody:

- an easy and intuitive language just as powerful as major competitors,
- open source, so anyone can contribute to its development,
- code that is as understandable as plain English,
- suitability for everyday tasks, allowing for short development times.

Rozwój języka jest prowadzony przy wykorzystaniu **PEP** (*Python Enhancement Proposal*). Dokumenty te to propozycje rozszerzeń lub zmian w języku w postaci artykułu, który jest poddawany pod dyskusję wśród programistów Pythona.

Każdy dokument zawiera opis proponowanego rozwiązania, uzasadnienie oraz aktualny status. Po osiągnięciu konsensusu propozycje są przyjmowane lub odrzucane.

Cechy Pythona:

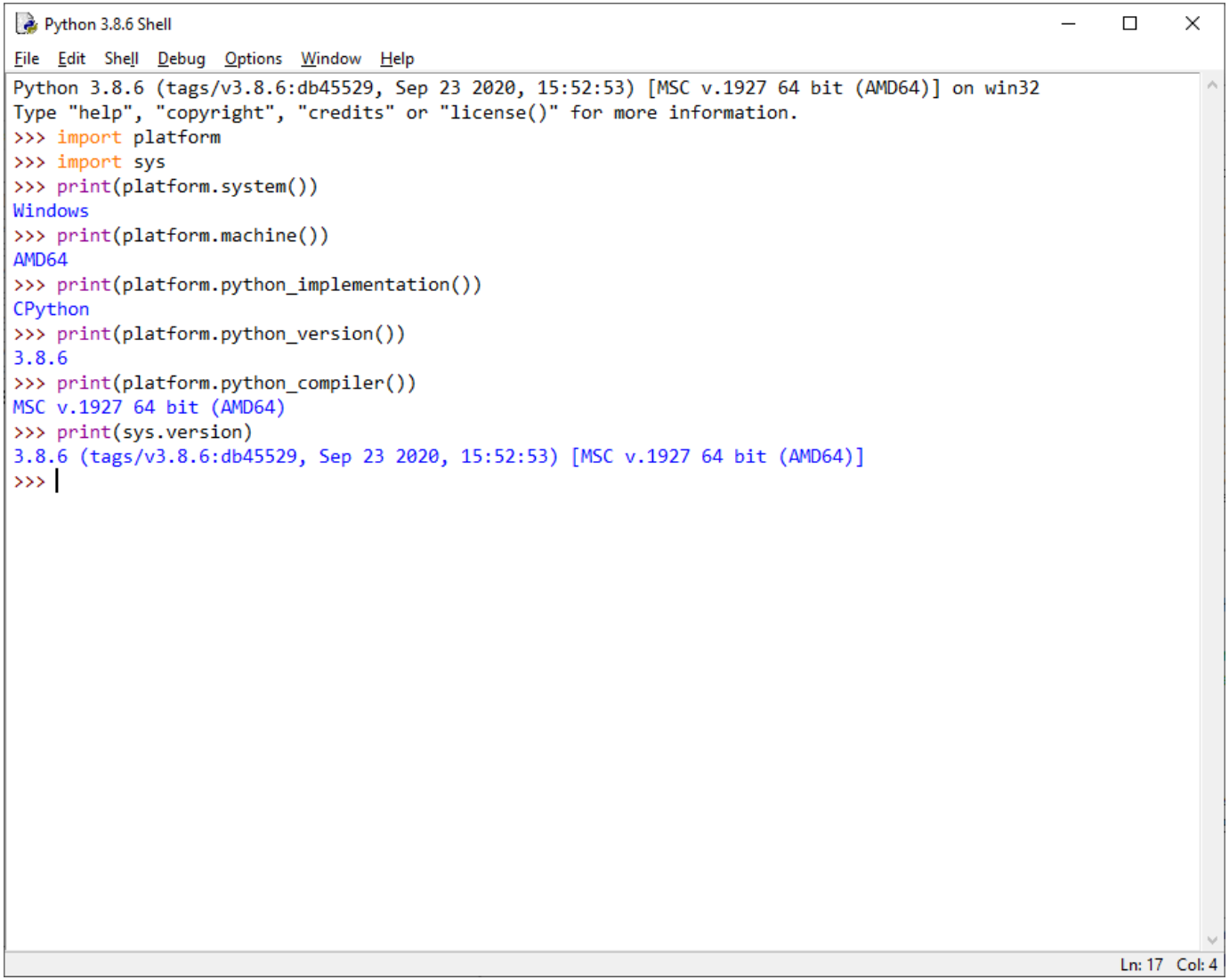
- jest językiem ogólnego przeznaczenia,
- jest w pełni obiektowy,
- umożliwia programowanie w różnych paragdygmatach (strukturalne, obiektowe, funkcyjne),
- jest dostępny dla różnych systemów operacyjnych,
- jest językiem o wzorcowej dokumentacji, dostępnej w sieci oraz lokalnie (po instalacji),
- jest językiem o dynamicznym typowaniu,
- jest językiem interpretowanym,
- chętnie jest stosowany do szybkiego prototypowania, automatyzacji i integracja komponentów, analizy danych.

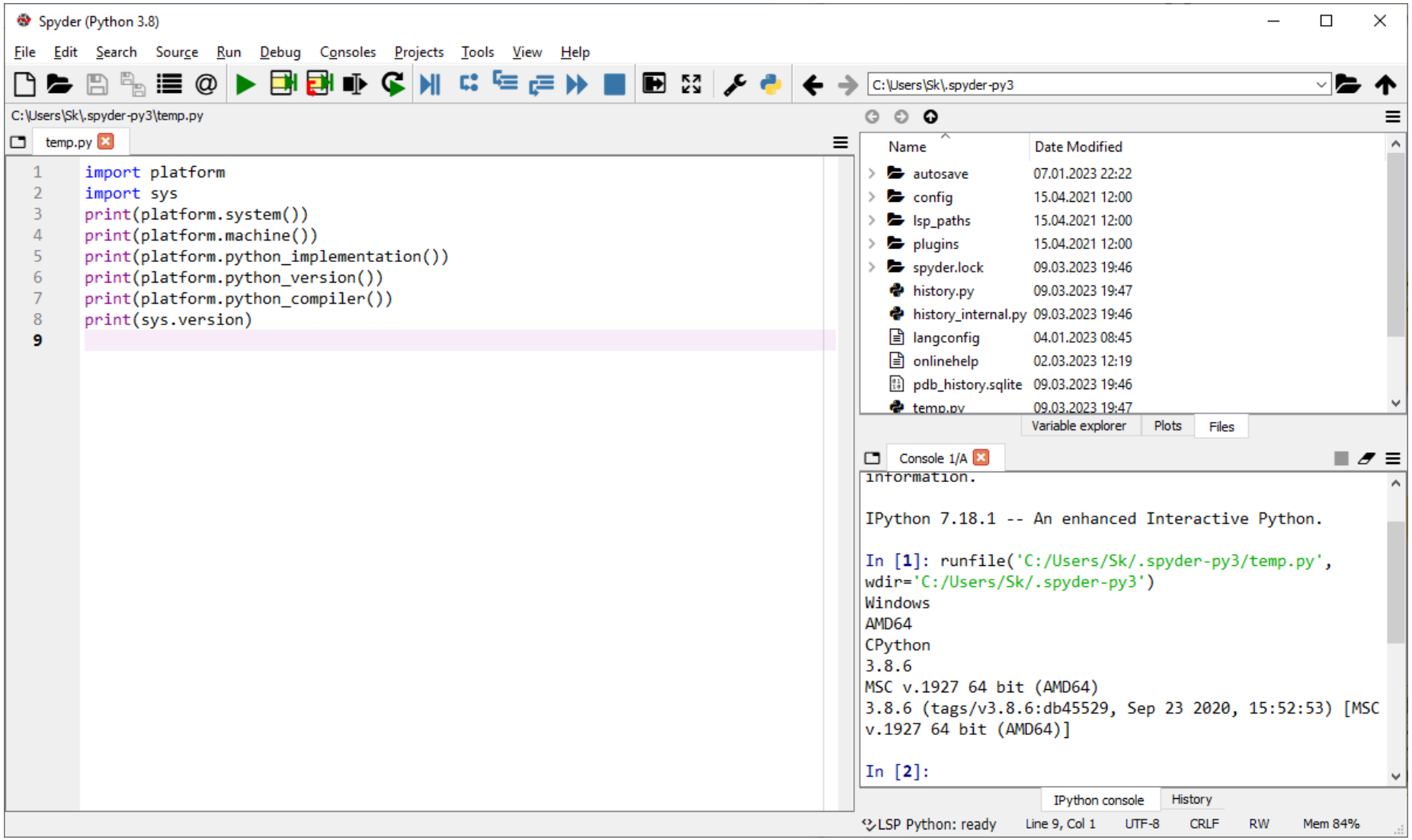
Implementacje kompilatora Pythona:

- **CPython** – implementacja referencyjna, kompilator napisany w języku C, najpopularniejszy, dystrybuowany z różnymi pakietami aplikacji i dołączany do różnych wersji SO,
- **PyPy** – kompilator napisany w Pythonie (RPythonie), skoncentrowany na wydajności, z kompilatorem JIT (*just in time*),
- **Jython** – kompilator skierowany na integrację z językiem programowania Java do, kod kompilowany do kodu bajtowego (*bytecode*) Javy, dla JVM,
- **IronPython** – kompilator zaprojektowany z myślą o umożliwieniu integracji programów napisanych w Pythonie z aplikacjami stworzonymi dla platformy .NET, kompilator do kodu bajtowego MSIL, dla .NET,
- **Numba** – kompilator do kodu maszynowego, wykorzystuje LLVM,
- **Cython** – kompilator/konwerter do C/C++.

Dwa sposoby wywołania interpretera:

- linia poleceń (konsola IDLE) – *read-eval-print loop* jak `bash` lub `PowerShell`,
- uruchomienie skryptu – polecenie `python foo.py`.




Spyder (Python 3.8)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\Sk\.spyder-py3
C:\Users\Sk\.spyder-py3\temp.py
temp.py
1 import platform
2 import sys
3 print(platform.system())
4 print(platform.machine())
5 print(platform.python_implementation())
6 print(platform.python_version())
7 print(platform.python_compiler())
8 print(sys.version)
9
Name
Date Modified
autosave 07.01.2023 22:22
config 15.04.2021 12:00
lsp_paths 15.04.2021 12:00
plugins 15.04.2021 12:00
spyder.lock 09.03.2023 19:46
history.py 09.03.2023 19:47
history_internal.py 09.03.2023 19:46
langconfig 04.01.2023 08:45
onlinehelp 02.03.2023 12:19
pdb_history.sqlite 09.03.2023 19:46
temp.py 09.03.2023 19:47
Variable explorer
Plots
Files
Console 1/A
information.
IPython 7.18.1 -- An enhanced Interactive Python.
In [1]: runfile('C:/Users/Sk/.spyder-py3/temp.py',
wdir='C:/Users/Sk/.spyder-py3')
Windows
AMD64
CPython
3.8.6
MSC v.1927 64 bit (AMD64)
3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC
v.1927 64 bit (AMD64)]
In [2]:
IPython console
History
LSP Python: ready
Line 9, Col 1
UTF-8
CRLF
RW
Mem 84%

Struktura programu

Python nie jest językiem o swobodnej składni – wcięcia w kodzie są wykorzystywane do tworzenia bloków (instrukcji złożonych).

Instrukcje jednakowo wcięte stanowią blok, nowy blok może pojawić się tylko w niektórych miejscach (zazwyczaj po dwukropku).

```
if True:
  foo = 10     ← Spaces
  print(foo)   ← Tab
```

Linia programu

Linia programu składa się z linii logicznych, a ta z kolei z linii fizycznych.

Linia fizyczna to sekwencja znaków zakończona kodem końca linii. W plikach źródłowych i ciągach znaków można zastosować dowolną ze standardowych sekwencji zakończenia linii platformy – ASCII **LF** (Unix/Linux), ASCII **CR LF** (Windows) lub ASCII **CR**.

Zalecana długość linii fizycznej: 79 znaków.

Zalecane wcięcie: 4 spacje (niestety zwykle stosuję dwie).

Kodowanie pliku źródłowego

Kod w podstawowej dystrybucji Pythona powinien zawsze używać **UTF-8** (lub **ASCII** w Pythonie 2).

Pliki używające ASCII lub UTF-8 nie powinny mieć deklaracji kodowania.

Wcięcia

```
→ Aligned with opening delimiter

foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

```
→ More indentation included to distinguish this from the rest

def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
```

```
→ Hanging indents should add a level

foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
```

```
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
```

```
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

Dzielenie linii z operatorami

```
→ Easy to match operators with operands

income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Puste linie

```
→ Surround top-level function definitions with two blank lines

def function_foo():
  return "foo has been done"


def function_bar():
  return "bar has been done"
```

Import

```
→ Imports should usually be on separate lines

import os
import sys
```

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

Białe znaki

→ Yes:
```
dct(foo[1], {bar: 2})
```

→ No:
```
dct( foo[ 1 ], { bar: 2 } )
```

→ Yes:
```
foo = (0,)
```

→ No:
```
bar = (0, )
```

→ Yes:
```
if x == 4: print x, y; x, y = y, x
```

→ No:
```
if x == 4 : print x , y ; x , y = y , x
```

→ Yes:
```
foo(1)
```

→ No:
```
foo (1)
```

→ Yes:
```
dct['key'] = lst[index]
```

→ No:
```
dct ['key'] = lst [index]
```

→ Yes:
```
foo[1:9], foo[1:9:3], foo[:9:3], foo[1::3], foo[1:9:]
foo[lower:upper], foo[lower:upper:], foo[lower::step]
foo[lower+offset : upper+offset]
foo[: upper_fn(x) : step_fn(x)], foo[:: step_fn(x)]
foo[lower + offset : upper + offset]
```

→ No:
```
foo[lower + offset:upper + offset]
foo[1: 9], foo[1 :9], foo[1:9 :3]
foo[lower : : upper]
foo[ : upper]
```

Konwencje nazewnictwa

- nazwy pakietów i modułów: `os`, `math`,
- nazwy klas: `decimal.Decimal`, `statistics.NormalDist`,
  - nazwy funkcji i zmiennych: `time.process_time`, `itertools.zip_longest`,
  - **niekonsekwentnie** np.: `re.findall`, `random.getrandbits`,
  - z jednym podkreśleniem na początku to zmienna niepubliczna np.: `_foo` (tylko konwencja),
  - z dwoma podkreśleniami na początku *name-mangling* np.: `__bar()` dostępna jako `_Foo__bar()`,
  - z dwoma podkreśleniami na początku do specjalnego użycia np.: `__init__`, `__str__`,
- nazwy stałych: `threading.TIMEOUT_MAX` (tylko konwencja),
  - zalecane definiowanie na poziomie modułu,
  - **niekonsekwentnie** np.: `math.pi`, `string.ascii_lowercase`.

Style nazewnictwa

Powszechnie wyróżnia się następujące style nazewnictwa:

- `b` (pojedyncza mała litera),
- `B` (pojedyncza duża litera),
- `lowercase`,
- `lower_case_with_underscores`,
- `UPPERCASE`,
- `UPPER_CASE_WITH_UNDERSCORES`,
- `CapitalizedWords`,
- `mixedCaseWords`,
- `Capitalized_Words_With_Underscores`.

Używając skrótów w **CapWords**, wszystkie litery skrótu należy pisać dużą literą. Lepiej `HTTPServerError` niż `HttpServerError`.

Z czego jest zbudowany program?

W trakcie procesu interpretowania lub kompilacji kod źródłowy jest dzielony na **jednostki leksykalne**. Jednostki leksykalne w Pythonie często są nazywane **tokenami**.

Rozróżnia się następujące klasy jednostek leksykalnych:

- identyfikatory (*identifiers*),
- słowa kluczowe (*keywords*),
- literały (*literals*),
- operatory (*operators*),
- separatory, ograniczniki (*punctuators*, *delimiters*).

Identyfikatory

Identyfikatory to ciągi liter, cyfr i znaków podkreślenia, muszą zaczynać się od litery lub znaku podkreślenia. Wielkie i małe litery są rozróżniane. Długość identyfikatorów jest nieograniczona.

Polskie znaki mogą, ale nie powinny być używane.

Identyfikatory są arbitralnie wybranymi nazwami dla **zmiennych**, **stałych**, **funkcji**, **typów danych** definiowanych przez programistę. Identyfikatory nie mogą być **słowami kluczowymi**.

```
maxvalue    mindelta    _5Pi
max_value   min_delta   _5_Pi
MaxValue    MinDelta    JW23
maxValue    minDelta    JW_23
```

Słowa kluczowe

Słowa kluczowe to identyfikatory zastrzeżone i nie mogą być inaczej stosowane niż określa to standard języka.

```
import keyword
print(keyword.kwlist)
```

Słowa kluczowe powinny być pisane tak jak je podano:

```
False      None       True       and          as         assert     async
await      break      class      continue     def        del        elif
else       except     finally    for          from       global     if
import     in         is         lambda       nonlocal   not        or
pass       raise      return     try          while      with       yield
```

Separatory

Cudzysłów (*double quote*) `"  "` – wykorzystywane do tworzenia napisów (łańcuchów znaków) jednowierszowych lub wielowierszowych `"""  """`, komentarze wielowierszowe.

Apostrofy (*single quote*) `'  '` – wykorzystywane do tworzenia napisów jednowierszowych lub wielowierszowych `'''  '''`.

Znak #, płotek, kratka (*hash*, *number sign*) `#` – komentarze.

Ukośnik wsteczny (*backslash*) `\` – używany do dzielenia długich linii, preferowanym sposobem jest użycie domniemanej kontynuacji wiersza w nawiasach i nawiasach klamrowych.

Znak at, tylko nie małpa (*at sign*) `@` – dekorator służący do modyfikowania definicji funkcji, metody lub klasy.

Nawiasy kwadratowe (*brackets*) `[  ]` – wykorzystywane są do deklarowania i odwoływania się do list.

Nawiasy okrągłe (*parentheses*) `(` `)` wykorzystywane są do grupowania wyrażeń, izolowania wyrażeń warunkowych, wskazują wywołanie funkcji i jej parametry, deklarowanie i odwoływanie się do krotek.

Nawiasy klamrowe (*braces*) `{` `}` – wykorzystywane są do deklarowania i odwoływania się do słowników.

Przecinek (*comma*) `,` rozdziela zwykle elementy na liście parametrów funkcji, elementy w listach i krotkach.

Średnik (*semicolon*) `;` – oddziela dwie instrukcje.

Dwukropek (*colon*) `:` – kończy nagłówek instrukcji złożonej.

Przypisanie (*equal sign*) `=` – przypisanie, stosowany pomiędzy nazwą a wartością przypisania.

Kropka (*dot*) `.` – daje dostęp do atrybutu obiektu.

Komentarze

Komentarze to fragmenty tekstu spełniające funkcje dowolnych objaśnień robionych przez programistów dla programistów.

Komentarze nie mogą występować w napisach i stałych znakowych. Komentarze są usuwane z tekstu źródłowego programu.

```
# To jest komentarz jednoliniowy

"""
Ten komentarz obejmuje
kilka linii
kodu
"""
```

Zmienne

Zmienna to konstrukcja programistyczna posiadająca trzy podstawowe atrybuty:

- symboliczną nazwę,
- miejsce przechowywania,
- wartość.

Zmienna pozwala w kodzie źródłowym na odwoływanie się przy pomocy nazwy do wartości lub miejsca przechowywania. Nazwa służy do identyfikowania zmiennej w związku z tym często nazywana jest **identyfikatorem**.

Miejsce przechowywania przeważnie znajduje się w pamięci komputera i określane jest przez adres i długość danych. Wartość to zawartość miejsca przechowywania.

Zmienna zazwyczaj posiada również czwarty atrybut: **typ**, określający rodzaj danych przechowywanych w zmiennej i co za tym idzie sposób reprezentacji wartości w miejscu przechowywania.

W programie wartość zmiennej może być odczytywana lub zastępowana nową wartością, tak więc wartość zmiennej może zmieniać się w trakcie wykonywania programu, natomiast dwa pierwsze atrybuty (nazwa i miejsce przechowywania) nie zmieniają się w trakcie istnienia zmiennej.

W językach ze **statycznym typowaniem** zmienna ma określony typ danych jakie może przechowywać. Jest on wykorzystywany do określenia reprezentacji wartości w pamięci, kontrolowania poprawności operacji wykonywanych na zmiennej (kontrola typów) oraz konwersji danych jednego typu na inny.

W językach z **typowaniem dynamicznym** typ nie jest atrybutem zmiennej lecz wartości w niej przechowywanej. Zmienna może wtedy w różnych momentach pracy programu przechowywać dane innego typu.

Deklaracja zmiennej to stwierdzenie, że dany identyfikator jest zmienną, przeważnie też określa typ zmiennej. W zależności od języka programowania deklaracja może być obligatoryjna, opcjonalna lub nie występować wcale.

Definicja oprócz tego, że deklaruje zmienną to przydziela jej pamięć. Podczas definiowania lub deklarowania zmiennej można określić jej dodatkowe atrybuty wpływające na sposób i miejsce alokacji, czas życia, zasięg i inne.

Zasięg zmiennej określa gdzie w treści programu zmienna może być wykorzystana, natomiast **czas życia** zmiennej to okresy w trakcie wykonywania programu gdy zmienna ma przydzieloną pamięć i posiada (niekoniecznie określoną) wartość.

Ze względu na zasięg można wyróżnić podstawowe typy zmiennych:

- **globalne** – obejmujące zasięgiem cały program,
- **lokalne** – o zasięgu obejmującym pewien blok, podprogram.

Podobnie ze zmiennymi w klasie mogą być dostępne:

- tylko dla danej klasy (zmienna **prywatna**),
- dla danej klasy i jej potomków (**zmienna chroniona**),
- w całym programie (**zmienna publiczna**).

Zmienne mogą zmieniać swój pierwotny zasięg na przykład poprzez importowanie/włącznie do zasięgu globalnego modułów, pakietów czy przestrzeni nazw.

Ze względu na czas życia i sposób alokacji zmienna może być:

- statyczna,
- automatyczna,
- dynamiczna.

Dla zmiennej **statycznej** pamięć jest rezerwowana w momencie kompilacji lub ładowania programu. Takimi zmiennymi są zmienne globalne, zmienne klasy (współdzielone przez wszystkie obiekty klasy, a nawet dostępne spoza klasy), statyczne zmienne lokalne funkcji (współdzielone pomiędzy poszczególnymi wywołaniami funkcji i zachowujące wartość po zakończeniu).

Zmiennej **automatycznej** pamięć jest automatycznie przydzielana w trakcie działania programu. Są to przeważnie zmienne lokalne podprogramów i ich parametry formalne, znikają po zakończeniu podprogramu.

Pamięć dla zmiennej **dynamicznej** alokowana jest ręcznie w trakcie wykonywania programu przy pomocy specjalnych konstrukcji lub funkcji. W zależności od języka zwalnianie pamięci może być ręczne lub automatyczne, Zazwyczaj nie posiada własnej nazwy, lecz odwoływać się do niej trzeba przy pomocy wskaźnika, referencji lub zmiennej o semantyce referencyjnej.

Typy zmiennych

Python jest językiem z dynamicznym typowaniem – przypisywanie typów do zmiennych odbywa się w trakcie wykonywania programu.

Przy samym uruchomieniu skryptu interpreter nie wie jeszcze jakiego typu będą zmienne. Określa to w momencie przypisania pierwszej wartości.

Typy w Pythonie są bardziej ogólne i mają większe możliwości, niż bywa to w innych językach. Na przykład: listy udostępniają uporządkowane zbiory różnych obiektów, słowniki przechowują obiekty według określonego klucza, obecność typu zespolonego. Zarówno listy, jak i słowniki mogą być zagnieżdżone, mogą zmieniać rozmiar. Typy są obiektowe.

Wybrane wbudowane typy zmiennych:

- logiczne `bool`,
- liczbowe:
  - całkowitoliczbowe `int`,
  - zmiennopozycyjne `float`,
  - zespolone `complex`,
- kolekcje (sekwencje):
  - listy `list`,
  - krotki `tuple`,
  - słowniki `dict`,
  - zbiory `set`,
- napisy lub łańcuchy znaków (sekwencja tekstowa) `str`.

Typ zmiennej jest ustalany na etapie wykonywania kodu i jest związany z literałem do niej przypisywanym lub z typami zmiennych biorących udział w instrukcji.

Na przykład, kiedy uruchomimy kod zawierający ciąg znaków ujęty w cudzysłów: `a = "foo"`, wykonujemy wyrażenie z literałem, które generuje i zwraca nowy obiekt typu `str`.

typ, literał	
liczby	`123`; `3.14`; `3+4j`; `0b111`; `Decinal()`; `Fraction()`
napisy	`'foo'`; `"bar"`; `b'a\x01c'`; `u'sp\xc4m'`
listy	`[1, [2, 'trzy'], 4.5]`; `list(range(10))`
słowniki	`{'fruit': 'apple', 'color': 'red'}`; `dict(hours=10)`
krotki	`(1,'foo', 4, 'U')`; `tuple('bar')`; `namedtuple`
pliki	`open('foo.txt')`; `open(r'C:\bar.bin', 'wb')`
zbiory	`set('abc')`; `{'a', 'b', 'c'}`

Sprawdzanie i porównywanie typów

```
a = c = 5
b = 5.0

print(type(a), type(b), a is b, a == b)
→ <class 'int'> <class 'float'> False True

print(type(a), type(c), a is c, a == c)
→ <class 'int'> <class 'int'> True True
```

```
print(type(a) is int)        ← niezalecane
print(isinstance(a, int)))   ← lepsze
```

Liczby

W Pythonie liczby nie są jednym typem obiektu, są raczej kategorią podobnych typów. Python obsługuje proste typy liczbowe (liczby całkowite oraz zmiennoprzecinkowe), a także literały służące do tworzenia liczb.

Dodatkowo Python udostępnia bardziej zaawansowaną obsługę programowania numerycznego, a także obiekty przeznaczone do bardziej zaawansowanych działań.

Możliwości języka obejmują:

- obiekty całkowite i zmiennoprzecinkowe,
- obiekty zespolone,
- obiekty dziesiętne o ustalonej precyzji,
- ułamkowe obiekty wymierne,
- zbiory: kolekcje z operacjami numerycznymi,
- wartości logiczne: `true` i `false`,
- wbudowane funkcje i moduły: `round`, `math`, `random` itp.,
- nieograniczoną precyzję liczb całkowitych,
- operacje bitowe,
- formaty szesnastkowe, ósemkowe i binarne,
- rozszerzenia tworzone przez niezależnych programistów: wektory, biblioteki, wizualizacje, tworzenie wykresów itp.

Typ logiczny

Wartości logiczne są przechowywane przez zmienne typu `bool`. Typ `bool` jest podklasą typu `int`.

Mogą one przyjmować wyłącznie wartości `True` lub `False`. Wartości logiczne są słowami kluczowymi.

Wartości logiczne prawie dokładnie odpowiadają wartościom `0` i `1`. Różnicę można zaobserwować w instrukcji `print(True)`.

Nie należy stosować porównań typu `foo == True`, ale samo `foo` lub `foo is True` (zmienna musi być typu `bool`).

Liczby całkowite

Liczby całkowite są przechowywane przez zmienne typu `int`. Typ `int`.

Rozmiar typu `int`

```
import sys
print(sys.getsizeof(int()))  ← 24
```

Od wersji Pythona 3.x liczby całkowite zwykłe oraz długie zostały połączone. Istnieje tylko jeden typ liczby całkowitej, który automatycznie obsługuje nieograniczoną precyzję.

Liczby całkowite mogą być w Pythonie zapisane jako **dziesiętne** (o podstawie dziesięć), **szesnastkowe** (o podstawie szesnaście), **ósemkowe** (o podstawie osiem) oraz **dwójkowe** (o podstawie dwa).

Literały szesnastkowe zaczynają się od znaków `0x` lub `0X`, po których następuje ciąg cyfr szesnastkowych od `0` do `9` i od `A` do `F`. W literałach szesnastkowych cyfry szesnastkowe mogą być zapisane małą i wielką literą.

Literały ósemkowe rozpoczynają się od znaków `0o` lub `0O` (cyfry zero i małej lub wielkiej litery **o**), po których następuje ciąg cyfr od `0` do `7`.

Literały dwójkowe rozpoczynają się od znaków `0b` lub `0B`, po których następują cyfry dwójkowe (`0` lub `1`).

Warto zauważyć, że wszystkie powyższe literały tworzą w kodzie programu obiekty liczb całkowitych, które są tylko alternatywnymi sposobami zapisu określonych wartości.

Wywołania wbudowanych funkcji `hex()`, `oct()` oraz `bin()` przekształcają liczbę całkowitą na jej **reprezentację** o podanej podstawie, natomiast wywołanie `int()` przekształca wykonywany łańcuch na liczbę całkowitą zgodnie z podaną podstawą.

Liczby rzeczywiste i zespolone

Liczby zmiennoprzecinkowe mają znak dziesiętny (w postaci kropki) lub zawierają opcjonalny wykładnik ze znakiem wprowadzony po literze `e` lub `E`, po której znajduje się opcjonalny znak.

Literał zapisany ze znakiem dziesiętnym lub wykładnikiem zostanie automatycznie zapisany jako obiekt typu `float`. Użycie liczby zmiennoprzecinkowej w wyrażeniu powoduje wykorzystanie arytmetyki liczb zmiennoprzecinkowych, a nie całkowitych.

Liczby zmiennoprzecinkowe w standardowych dystrybucjach CPython zaimplementowane są jako typ `double` z języka C, dlatego mają taką precyzję, jaką kompilator języka C wykorzystany do zbudowania interpretera Pythona przydzieli liczbom tego typu.

Literały zespolone mają postać $a + b \cdot \mathrm{j}$, gdzie a to część **rzeczywista**, b to część **urojona** liczby, a j to jednostka urojona $\mathrm{j}^2 = -1$.

Jeżeli `z` jest zmienną zespoloną, to `z.real` i `z.imag` to odpowiednio część rzeczywista i urojona tej liczby.

literał, interpretacja	
`1234`; `-24`; `0`; `99999999999999`	liczby całkowite o nieograniczonej wielkości (*integer*)
`1.23`; `1.`; `3.14e-10`; `4E21`	liczby zmiennoprzecinkowe (*floating-point*)
`0o177`; `0x9ff`; `0b101010`	literały ósemkowe, szesnastkowe i dwójkowe
`3+4j`; `3.0+4.0j`; `3J`	liczby zespolone (*complex*)
`set('foo')`; `{1, 2, 3, 4}`	zbiory
`Decimal('1.0')`; `Fraction(1,3)`	typy rozszerzające: dziesiętne i ułamkowe
`bool(foo)`; `True`; `False`	typ logiczny i stałe

Przykłady formatowania liczb

```
a = 3
b = 4
print(b / (2 + a))  ← 0.8

foo = 1 / 3
bar = 1 / 6
num = 5

print(foo)  ← 0.333333333333333
print("foo = %e" % foo)  ← 3.333333e-01
print("foo = %4.2f" % foo)  ← 0.33

print("foo = {}; bar = {}" . format(foo, bar))
→ foo = 0.3333333333333333; bar = 0.16666666666666666

print("bar = {1}; foo = {0}" . format(foo, bar))
→ bar = 0.16666666666666666; foo = 0.3333333333333333

print("foo = {0:4.2f}; bar = {1:4.4f}; num = {2:3d}"
    format(foo, bar, num))
→ foo = 0.33; bar = 0.1667; num =   5
```

Napisy (łańcuchy znaków)

Z funkcjonalnego punktu widzenia łańcuchy znaków można wykorzystać do reprezentowania wszystkiego, co można zakodować w postaci tekstowej lub bajtowej.

W dziedzinie tekstów obejmuje to między innymi symbole i słowa, pliki tekstowe załadowane do pamięci, adresy internetowe czy kod źródłowy programów.

Ciągów znaków można również używać do przechowywania surowych danych bajtowych używanych na przykład w plikach multimedialnych i transferach sieciowych, a także zakodowanych i zdekodowanych tekstów w formacie Unicode.

Napisy w Pythonie należą do typu `str`. Literały tego typu można tworzyć na kilka sposobów:

- napisy mieszczące się w jednej linii można umieścić wewnątrz **cudzysłowów** (zalecane) lub **apostrofów**,
- napisy wielowierszowe umieszcza się wewnątrz potrójnych cudzysłowów lub apostrofów,
- poprzedzenie napisu literą `r` spowoduje, że znaki specjalne (takie jak `n`) staną się zwykłymi znakami.

Napisy są obiektami niemodyfikowalnymi. Każda operacja modyfikująca napis powoduje stworzenie nowego obiektu. W szczególności nie należy zapętlać operacji z wykorzystaniem `+=`.

Od Pythona 3.3 napis ma reprezentację znaków o długości 1, 2 lub 4 bajty, ustalaną w czasie uruchomienia według wartości znaków.

Aby wymusić stworzenie ciągu znaków **ASCII**, trzeba poprzedzić napis literą `b`.

Funkcje `ord()` i `chr()` służą do konwersji znaku na wartość kodu i zmiany kodu na znak.

Znaki diakrytyczne różnych języków i znaki specjalne są wspierane na wiele sposobów:

- zakodowanie znaku bezpośrednio np.: ą,
- zakodowanie znaku poprzez kod Unicode np.: `\u0105`,
- zakodowanie znaku poprzez nazwę Unicode np.: `\N{LATIN SMALL LETTER A WITH OGONEK}`,
- zakodowanie znaku poprzez łączenie nazw Unicode np.: `\N{LATIN SMALL LETTER A}\N{COMBINING OGONEK}`.

literały, operacje na napisach

`S = ''`	pusty łańcuch znaków
`S = "zmienna 'foo'"`	cudzysłowy
`S = 'f\no\to\x00bar'`	sekwencje znaków specjalnych
`S = """...wiele wierszy..."""`	Blok w potrójnych cudzysłowach
`S = r'\temp\foo'`	surowy łańcuch znaków (bez znaków ucieczki)
`S = b'bar'`	bajtowe łańcuchy znaków
`S = u'foo'`	łańcuch znaków Unicode
`S + S`; `S * 3`	konkatenacja i powtórzenie
`S[i]`; `S[i:j]`; `len(S)`	indeksowanie, wycinek, długość
`S.find('foo')`	wyszukiwanie
`S.rstrip()`	usuwanie białych znaków
`S.replace('foo', 'bar')`	zastępowanie
`S.split(',')`	dzielenie w miejscu wystąpienia separatora
`S.lower()`	konwersja wielkości liter

Sekwencje ucieczki (sekwencje specjalne) pozwalają osadzać w łańcuchach znaki, których nie da się łatwo wpisać z klawiatury.

Znak `\` i jeden lub większa liczba następujących po nim znaków w literale łańcuchowym w wynikowym obiekcie łańcucha znaków zastępowane są pojedynczym znakiem o wartości binarnej określonej przez sekwencję ucieczki.

sekwencje specjalne, znaczenie	
`\\`	ukośnik wsteczny (czasem lewy)
`\'`	apostrof
`\"`	cudzysłów
`\a`	sygnał dźwiekowy
`\b`	znak cofania (*backspace*)
`\f`	wysunięcie strony (*form feed*)
`\n`	nowy wiersz (*line feed*)
`\r`	powrót karetki
`\t`	tabulator poziomy
`\v`	tabulator pionowy

Listy

Obiekt reprezentujący listę jest ogólnym rodzajem **sekwencji**. Listy mają typ `list`.

Listy są uporządkowanymi kolekcjami dowolnych obiektów. Zachowują uporządkowanie elementów (są sekwencjami).

Dostęp do elementów list można uzyskać za pomocą pozycji przesunięcia. Element listy można z niej pobrać, indeksując listę w pozycji przesunięcia obiektu. Ponieważ elementy listy są uporządkowane pozycyjnie, możliwe jest wykonywanie wycinków czy konkatenacja.

Listy mają zmienną długość. Mogą również zawierać dowolny typ obiektów, są zatem niejednorodne (heterogeniczne). Ponieważ listy mogą zawierać inne obiekty, obsługują również dowolne zagnieżdżanie. Możliwe jest tworzenie list składających się z innych list.

Listy należą do sekwencji mutowalnych, można modyfikować je w miejscu i reagują na wszystkie operacje na sekwencjach, takie jak indeksowanie, wycinki i konkatenacja. Operacje takie po zastosowaniu do list zwracają nowe listy. Listy obsługują również operacje usuwania czy przypisania do indeksu.

Listy są tablicami referencji do obiektów, zawierają zero lub większą liczbę referencji do innych obiektów. Listy przypominają tablice wskaźników (adresów), dlatego pobranie elementu z listy jest bardzo szybkie.

Przypisanie obiektu do komponentu struktury danych czy nazwy zmiennej, spowoduje przechowanie referencji do samego obiektu, a nie jego kopię (o ile tego w jawny sposób nie zażądamy).

literały, operacje na listach

`L = []`	pusta lista
`L = [123, 'abc', 1.23, ]`	cztery elementy, indeksy od 0 do 3
`L = ['Adam', 40.0, ['prog', 'python']]`	zagnieżdżone podlisty
`L = list('mielonka')`	lista elementów obiektu iterowanego
`L = list(range(-4, 4))`	lista kolejnych liczb całkowitych
`L[i]; L[i][j]; L[i:j]; len(L)`	indeks, indeks indeksu, wycinek, długość
`L + L; L * 3`	konkatenacja, powtórzenie
`for x in L:print(x); 3 in L`	iteracja, przynależność
`L.append(4); L.extend([5, 6, 7])`	dodawanie elementów
`L.insert(i, X)`	
`L.index(X); L.count(X)`	przeszukiwanie
`L.sort(); L.reverse(); L.copy()`	sortowanie, odwracanie, kopiowanie
`L.clear()`	czyszczenie
`L.pop(i); L.remove(X); del L[i]`	zmniejszanie listy
`del L[i:j]; L[i:j] = []`	
`L[i] = 3; L[i:j] = [4, 5, 6]`	przypisanie do indeksu i wycinka
`L = [x**2 for x in range(5)]`	listy składane i odwzorowania
`List(map(ord, 'foobar'))`	

Jeśli chcemy efektywnie przechować dane tylko jednego typu, możemy użyć tablicy.

```
import array

tab = array.array('d', [1, 2, 3, 4])
```

Alternatywnie można użycie klasy `numpy.array`. W praktyce dużo lepsze dla obliczeń matematycznych, zawiera wbudowaną wektoryzację wielu operacji.

Słowniki

Słowniki to struktury danych, które przechowują pary **klucz-wartość**. Klucze muszą być obiektami niemodyfikowalnymi. Słowniki należą do typu `dict`.

Słowniki nie są sekwencjami, ale są określane jako **odwzorowania** (*mappings*). Odwzorowania są również zbiorami innych obiektów, w słownikach obiekty są przechowywane według klucza, a nie ich pozycji względnej.

Słowniki, mogą rozszerzać się i kurczyć w miejscu (bez tworzenia nowych kopii) i mogą zawierać obiekty dowolnego typu. Obsługują zagnieżdżanie na dowolną głębokość (mogą zawierać listy, inne słowniki itp.).

Podobnie jak listy, są elastycznym narzędziem do reprezentowania kolekcji, ale ich bardziej mnemoniczne klucze lepiej nadają się do zastosowania, kiedy elementy kolekcji są nazwane lub oznaczone.

Każdy klucz może mieć tylko jedną powiązaną wartość, ale taka wartość może być zbiorem wielu obiektów, a każdą wartość można zapisać pod dowolną liczbą kluczy.

Słowniki dobrze zastępują rekordy, tabele wyszukiwania i wszelkie inne rodzaje agregacji, w których nazwy przedmiotów są bardziej znaczące niż ich pozycje.

Do pobrania komponentów słownika wykorzystuje się tę samą operację indeksowania co w przypadku list, jednak indeks ma tutaj postać klucza, a nie względnej wartości przesunięcia.

Python wykorzystuje zoptymalizowane algorytmy haszujące, służące do odnajdywania kluczy, dzięki czemu to pobieranie jest szybkie.

Podobnie jak listy, tak i słowniki przechowują referencje do obiektów (a nie ich kopie, o ile nie zdefiniuje się tego w sposób jawny).

literały, operacje na słownikach

`D = {}`	pusty słownik
`D = {'imie': 'Adam', 'wiek': 33}`	słownik dwuelementowy
`E = {'cto': {'imie': 'Adam', 'wiek': 33}}`	zagnieżdżanie
`D = dict(imie='Adam', wiek=33)`	inne techniki tworzenia: słowa kluczowe,
`D = dict([('imie', 'Adam'), ('wiek', 33)])`	pary klucz-wartość, spięte pary
`D = dict(zip(keyslist, valueslist))`	listy kluczy
`D = dict.fromkeys(['imie', 'wiek'])`	
`D['imie']; D['cto']['wiek']`	indeksowanie po kluczu
`'wiek' in D`	sprawdzanie przynależności klucza
`D.keys()`	wszystkie klucze
`D.values()`	wszystkie wartości
`D.items()`	wszystkie krotki klucz+wartość
`D.copy()`	kopiowanie
`D.clear()`	czyszczenie (usuwa wszystkie elementy)

literały, operacje na słownikach

`D.update(D2)`	łączenie według kluczy
`D.get(key, default?)`	pobieranie według klucza, wartości domyślne (lub `None`), gdy brak klucza
`D.pop(key, default?)`	usuwanie według klucza, wartości domyślne (lub błąd), gdy brak klucza
`D.setdefault(key, default?)`	pobieranie według klucza, wartości domyślne (lub `None`), gdy brak klucza
`D.popitem()`	usuwanie lub zwracanie pary (klucz, wartość)
`len(D)`	długość (liczba przechowywanych wpisów)
`D[key] = 42`	dodawanie lub modyfikacja kluczy
`del D[key]`	usuwanie elementu według klucza
`list(D.keys())`	widoki słowników
`D1.keys() & D2.keys()`	
`D = {x: x*2 for x in range(10)}`	słowniki składane

Krotki

Krotki (pary, trójki) to listy o stałym rozmiarze, których nie można modyfikować. Krotki należą do typu `tuple`.

Krotki są uporządkowanymi kolekcjami obiektów — zachowują zatem kolejność elementów od lewej do prawej strony. Tak jak w przypadku list, również w krotkach możemy osadzać dowolne typy obiektów.

Dostęp do elementów krotki odbywa się za pomocą wartości przesunięcia (a nie według klucza). Krotki obsługują wszystkie operacje oparte na wartościach przesunięcia, w tym indeksowanie i wycinki.

Krotki są sekwencjami — obsługują wiele z operacji odnoszących się do typów sekwencyjnych. Są jednak niemutowalne, stąd nie obsługują żadnych operacji modyfikujących w miejscu, które mają zastosowanie do list.

Krotki mają stałą długość, są heterogeniczne i można je dowolnie zagnieżdżać. Ponieważ krotki są niemutowalne, nie można zmieniać ich rozmiaru bez tworzenia kopii.

Krotki mogą przechowywać dowolne typy obiektów, w tym inne obiekty złożone (na przykład listy, słowniki i inne krotki) i mogą być praktycznie dowolnie zagnieżdżone.

Krotki są tablicami referencji do obiektów. Krotki przechowują punkty dostępu do innych obiektów (referencje), a indeksowanie krotek jest dość szybkie.

literały, operacje na krotkach

`()`	pusta krotka
`T = (0,)`	krotka jednoelementowa (nie jest wyrażeniem)
`T = (0, 'Ni', 1.2, 3)`	krotka czteroelementowa
`T = 0, 'Ni', 1.2, 3`	inna krotka czteroelementowa (ta sama co wyżej)
`T = ('Adam', ('prog', 'pyton'))`	zagnieżdżone krotki
`T = tuple('foo')`	krotka elementów w obiekcie iterowalnym
`T[i]`; `T[i][j]`; `T[i:j]`; `len(T)`	indeks, indeks indeksu, wycinek, długość
`T + T`; `T * 3`	konkatenacja, powtórzenie
`for x in T: print(x)`	iteracja, przynależność
`'foo' in T`	
`[x ** 2 for x in T]`	
`T.index('Ni')`; `T.count('Ni')`	wyszukiwanie, zliczanie
`namedtuple('Emp', ['name', 'jobs'])`	krotka z nazwanymi polami (krotka typu named tuple)

Operator w programowaniu konstrukcja językowa jednoargumentowa, bądź wieloargumentowa zwracająca wartość.

Do podstawowych operatorów, będących elementem większości języków programowania, należą operatory: **przypisania**, **arytmetyczne**, **relacji** (porównania), **logicznie**.

Główne cechy opisujące operator to:

- liczba i typy argumentów,
- typ wartości zwracanej,
- wykonywane działanie,
- priorytet,
- łączność lub jej brak,
- umiejscowienie operatora względem operandów.

Przypisanie

Przypisanie (podstawienie) jest to operacja nadania, umieszczenia, wpisania do określonej **L-wartości** (jest to wartość, która istnieje dłużej niż przez jedno wyrażenie i można pobrać jej adres, jest nią też zmienna) nowej wartości.

Przypisanie może zostać dokonane:

- instrukcją przypisania,
- operatorem przypisania,
- innym operatorem,
- w inicjalizacji zmiennej,
- w wywołaniu podprogramu,
- w wyniku efektów ubocznych,
- w instrukcji wejścia.

Operator przypisania to jeden z podstawowych operatorów prostych występujących w językach programowania.

Zwykle nie jest słowem kluczowym, choć istnieją języki programowania wymagające lub zezwalające opcjonalnie na użycie słowa kluczowego.

W Pythonie operatorem przypisania jest znak równości =. Operator przypisania występuje w **instrukcji przypisania**.

Operator przypisania wartościuje listę wyrażeń (to może być pojedyncze wyrażenie lub lista wyrażeń rozdzielonych przecinkami, gdzie drugi przypadek reprezentuje krotkę) i przypisuje każdemu z elementów z listy docelowej pojedynczy obiekt wynikowy, w kolejności od lewej do prawej.

Przypisania tworzą referencje do obiektów. Zmienne tworzone są przy pierwszym przypisaniu. Przed odwołaniem się do zmiennej trzeba ją najpierw przypisać. Niektóre operacje wykonują przypisania niejawne.

Operator przypisania w Pythonie jest **prawostronnie łączny**. Umożliwia to łączenie przypisań:

```
i = 5

l = k = j = i;
```

operacja, interpretacja	
`foo = 'bar' i = 9`	forma podstawowa
`foo, bar = 'ala', 'pies'`	przypisanie krotki (pozycyjne)
`[foo, bar] = ['ala', 'pies']`	przypisanie listy (pozycyjne)
`a, b, c, d = 'foo'`	przypisanie sekwencji, uogólnione
`a, *b = 'bar'`	rozszerzone rozpakowanie sekwencji
`foo = bar = 'var'`	przypisanie do wielu celów
`foo += 42`	przypisanie rozszerzone (`foo = foo + 42`)

Operatory arytmetyczne

Operator arytmetyczny to operator, który działając na podanych argumentach reprezentujących wartości liczbowe, w wyniku zwraca również wyznaczoną wartość liczbową, realizując podstawowe operacje arytmetyki.

To jakie operatory arytmetyczne są dostępne w konkretnym języku programowania zależy od jego składni, a to jakie są zasady ich stosowania, w tym priorytet tych operatorów i kolejność opracowywania argumentów, od przyjętej implementacji języka.

Zróżnicowany jest również sposób zapisu operatorów arytmetycznych. Stosuje się zapis, bądź za pomocą symboli (znaku lub znaków nie będących literami), w konwencji zbliżonej do matematycznej, bądź zdecydowanie rzadziej w postaci słów kluczowych.

Operatory arytmetyczne realizują następujące operacje arytmetyczne (przykłady):

- jednoargumentowe:
  - zmiana znaku liczby (wyznaczenie liczby przeciwnej),
  - zachowanie znaku liczby,
  - inkrementacja,
  - dekrementacja,
- dwuargumentowe:
  - dodawanie,
  - odejmowanie,
  - mnożenie,
  - dzielenie,
  - dzielenie całkowitoliczbowe,
  - reszta z dzielenia całkowitoliczbowego,
  - potęgowanie.

operator, opis	
`x + y`	dodawanie, konkatenacja
`x - y`	odejmowanie, różnica zbiorów
`x * y`	mnożenie, powtórzenie
`x % y;`	reszta z dzielenia, format
`x / y; x // y;`	dzielenie, dzielenie bez reszty
`x ** y`	potęga
`-x; +x`	negacja, identyczność

Wyrażenie arytmetyczne jest to językach programowania dowolne wyrażenie typu liczbowego. Może być ono złożone ze zmiennych, liczb, funkcji, symboli działań (tu: operatorów), itp.

```
result =  10  * 3  ← 30
result =  10  / 3  ← 3
result =  10  % 3  ← 1
result =  10 ** 2  ← 100
result =  10  + 3  ← 13
result =  10  - 3  ← 7
result =  5        ← 5
result = -result   ← -5
```

```
result = 10.0 / 3.0  ← 3.333333
result = 10 / 3      ← 3 -- wynik możne zależeć od typów literałów
                        (C, C++, Python 2.x)
```

Kolejność (priorytety) wykonywania działań jest taka jak w matematyce, można ją regulować za pomocą nawiasów okrągłych. Operatory równoważne wykonywane są od strony lewej do prawej (należy wziąć pod uwagę wiązanie).

wybrane funkcje matematyczne z `math`	
`sin()`	sinus
`cos()`	cosinus
`tan()`	tangens
`asin()`	arcus sinus
`acos()`	arcus cosinus
`atan()`	arcus tangens
`exp()`	eksponent (e^x)
`log()`	logarytm naturalny
`log10()`	logarytm dziesiętny
`pow()`	potęga
`sqrt()`	pierwiastek kwadratowy
`ceil()`	zaokrąglenie do liczby całkowitej w górę
`floor()`	zaokrąglenie do liczby całkowitej w dół
`fabs()`	wartość bezwzględna

Operatory relacji i wyrażenia logiczne

Operator **relacji** jest to operator który działając na podanych argumentach, w wyniku zwraca wartość logiczną, określającą spełnienie bądź nie spełnienie reprezentowanej przez ten operator relacji zachodzącej między argumentami.

Wynikiem działania operatora relacji jest więc wartość reprezentująca zgodnie z zasadami obowiązującymi w składni języka programowania jedną z wartości logicznych: **prawdę** (*true*) lub **fałsz** (*false*).

W językach programowania dostępne są operatory, które badają relacje:

- równości,
- nierówności,
  - negacji równości,
  - nierówności ostrych,
    - mniejsze,
    - większe,
  - nierówności nieostrych,
    - mniejsze lub równe,
    - większe lub równe,
- przynależności (zawierania),
- równoważności.

operatory relacji	
==	czy równe?
!=	czy różne?
>	czy większe?
<	czy mniejsze?
>=	czy większe lub równe?
<=	czy mniejsze lub równe?

Operatory relacji mogą działać na wszystkich typach danych, jednak najczęściej odnoszą się do danych numerycznych.

Porównanie danych całkowitych nie powinno budzić zastrzeżeń, ponieważ operacje wykonywane są w sposób naturalny.

Należy zachować ostrożność przy porównywaniu wartości zmiennoprzecinkowych.

```
first = 1.00000000000001
second = 1.00000000000002

if(first == second)
  ...
```

```
first = 1.00000000000001
second = 1.00000000000002

if(fabs(first - second) < 0.000001)
  ...
```

Operator **logiczny** to operator, który działając na argumentach reprezentujących wartości logiczne, w wyniku zwraca również wartość logiczną, realizując podstawowe operacje algebry Boole'a.

Dostępność operatorów logicznych, priorytet i kolejność opracowywania argumentów zależy od przyjętej implementacji języka.

Zróżnicowany jest również sposób zapisu operatorów logicznych, stosuje się zapis, bądź w postaci słów kluczowych, bądź symboli (znaku lub znaków nie będących literami).

Operatory logiczne realizują następujące operacje logiczne:

- jednoargumentowe:
  - negacja,
- dwuargumentowe:
  - koniunkcja,
  - alternatywa,
  - alternatywa wykluczająca,
  - implikacja,
  - ekwiwalencja.

operatory logiczne	
`not`	negacja (zaprzeczenie)
`and`	koniunkcja
`or`	alternatywa

`not`, nargument, wynik	
prawda	fałsz
fałsz	prawda

`and`, argumenty, wynik		
prawda	prawda	prawda
prawda	fałsz	fałsz
fałsz	prawda	fałsz
fałsz	fałsz	fałsz

`or`, argumenty, wynik		
prawda	prawda	prawda
prawda	fałsz	prawda
fałsz	prawda	prawda
fałsz	fałsz	fałsz

Wyrażenie logiczne jest to językach programowania dowolne wyrażenie zawierające operatory relacji i operatory logiczne, stałe, zmienne logiczne, którego wynik jest typu logicznego (prawda lub fałsz).

Wyrażenia logiczne mogą zawierać wyrażenia innego typu, np. arytmetyczne.

```
result = (5 < 3 * 2) and ('L' > 'A')
result = (5 != 3 * 2) or ('L' == 'A')
result = not(2 + 2 == 5)
```

```
value = 25
even = (value % 2) == 0
extent = ((value >= 10) and (value <= 20))
result = even and extent
```

Operatory identyczności pozwalają określić, czy dwie zmienne przechowują ten sam obiekt. Występują w dwóch wersjach: `is` i `not is`.

```
x = "ala ma psa"
y = "ala nie ma psa"

if x is not y:
  print("obiekty nie są takie same")

x = y

if x is y:
  print("obiekty są takie same")
```

Operatorem `is` (`not is`) można także sprawdzić, czy dany obiekt jest oczekiwanym typem.

```
x = "ala ma psa"
y = 25

if type(x) is str:
  print("obiekt x jest typu str")

if type(y) is int:
  print("obiekt x jest typu int")
```

Operatory przynależności tego typu sprawdzają, czy dany element zawiera się w podzbiorze wartości danego obiektu (po których można iterować). Występują w dwóch wersjach: `in` i `not in`.

```
x = "ala ma psa"

if "ma" in x:
  print("wyraz 'ma' występuje w ciągu")

y = [2, 3, 4, 66]

if 4 in y:
  print("liczba 4 występuje w zbiorze")
else:
  print("liczba 4 nie występuje w zbiorze")
```

Dla większości operatorów dwuargumentowych można wykorzystać specjalne operatory przypisania (*shorthands*), pozwalające skrócić zapis, na przykład:

- mnożenie: `a *= b` zamiast `a = a * b`,
- dzielenie: `a /= b` zamiast `a = a / b`,
- reszta z dzielenia całkowitoliczbowego: `a %= b` zamiast `a = a % b`,
- dodawanie: `a += b` zamiast `a = a + b`,
- odejmowanie: `a -= b` zamiast `a = a - b`,

Operator warunkowy

Często spotyka się *symetryczne* instrukcje warunkowe:

```
if(delta > 0):
  isSolution = 1
else:
  isSolution = 0
```

```
if(a > b):
  max = a
else:
  max = b
```

Można je zapisać krócej z wykorzystaniem **operatora warunkowego** (operatora trójargumentowego):

```
isSolution = 1 if(delta > 0) else 0
```

```
max = a if(a > b) else b
```

Instrukcja warunkowa

Instrukcja warunkowa jest elementem języka programowania, który pozwala na wykonanie różnych obliczeń (zadań) w zależności od tego czy zdefiniowane przez programistę **wyrażenie logiczne** jest prawdziwe, czy fałszywe.

```
a = 1
b = 2

if(a + b == 3):

  print("suma jest równa 3")
```

```
take = 29999.99

if(take >= 30000):

  bonus = 0.05 * take
  print("Premia w wysokości: %f" % bonus)

else:

  bonus = 0
  print("Premii nie będzie!")
```

```
a = 1
b = 8
c = 2

delta = b * b - 4 * a * c

if(delta > 0):

  x1 = (-b - math.sqrt(delta)) / (2 * a)
  x2 = (-b + math.sqrt(delta)) / (2 * a)

  print("x1 = %f; x2 = %f" %(x1, x2))

elif(delta == 0):

  x0 = -b / (2 * a)
  print("x0 = %f" % x0)

else:

  print("Brak pierwiastków rzeczywistych")
```

Instrukcja wyboru

Instrukcja wyboru jest instrukcją umożliwiającą wybór instrukcji do wykonania spośród wielu opcji.

Składnia instrukcji wyboru różni się w zależności od języka programowania, lecz można wyróżnić w niej charakterystyczne elementy:

- nagłówek instrukcji wyboru – słowo kluczowe rozpoczynające instrukcję, nazwa zmiennej lub wyrażenie, na podstawie którego następuje wybór,
- ciało instrukcji wyboru – kolejne instrukcje (bloki instrukcji) podlegające selekcji, poprzedzone frazami zawierającymi wartości, listy lub zakresy porównywane z wyrażeniem (zmienną) z nagłówka instrukcji,
- opcjonalnie fraza domyślna, wykonywana gdy żadna z fraz nie spełni warunku,
- koniec instrukcji wyboru.

Instrukcja wyboru (zwykle `switch(expression)` ... `case constant: statement`) nie występuje w Pythonie. Można ją zastąpić wielokrotną instrukcją wyboru.

Pętle

Pętla to jedna z podstawowych konstrukcji programowania strukturalnego (obok instrukcji warunkowej i instrukcji wyboru). Umożliwia cykliczne wykonywanie ciągu instrukcji:

- określoną liczbę razy (pętla **iteracyjna**, licznikowa),
- do momentu zajścia pewnych warunków (pętla **repetycyjna**, warunkowa),
- dla każdego elementu kolekcji (pętla **foreach**, po kolekcji),
- w nieskończoność.

Pętla iteracyjna, to rodzaj pętli, w której następuje wykonanie określonej liczby iteracji. Do kontroli przebiegu wykonania pętli iteracyjnej stosuje się specjalną zmienną, którż nazywa się **zmienną sterująca**, **kontrolną** lub **licznikową**.

W ramach pętli przejście do kolejnej iteracji wiąże się ze zmianą wartości zmiennej sterującej o określoną wielkość i sprawdzenie warunku, czy nowa wartość zmiennej sterującej znajduje się nadal w dopuszczalnym zakresie wartości, określonym dla tej zmiennej.

Kolejność wykonywania działań jest następująca:

- przypisanie wartości początkowej do zmiennej sterującej,
- sprawdzenie, czy wartość zmiennej sterującej mieści się w dopuszczalnym zakresie wartości, tzn. jej wartość jest równa lub mniejsza od wartości granicznej, albo jest równa lub większa od wartości granicznej:
  - jeżeli wartość zmiennej sterującej nie mieści się w dopuszczalnym zakresie wartości to kończy wykonywanie pętli,
  - jeżeli wartość zmiennej sterującej mieści się w dopuszczalnym zakresie wartości to nie przerywa działania,
- wykonuje iterację,
- zmienia wartość zmiennej sterującej o zadany krok, wartość ta może być zwiększana lub zmieszana.

Zmienna sterująca służy do kontroli przebiegu realizacji pętli iteracyjnej. Przyjmuje ona kolejne wartości z zadanego zakresu zmieniane o określoną wartość kroku.

W różnych językach programowania mogą być stawiane określone wymagania i ograniczenia dotyczące zmiennych sterujących. Ograniczenia te mogą dotyczyć takich atrybutów tej zmiennej jak np. dozwolony typ danych, zasięgu.

Zmiana wartości zmiennej sterującej może nastąpić:

- automatycznie, w sposób ukryty,
- jawnie, w kodzie bloku pętli, o ile dany język programowania dopuszcza taką konstrukcję.

Ogólniejszą konstrukcją jest pętla **repetycyjna** (warunkowa), która jest wykonywana, aż do odpowiedniej zmiany warunków.

Warunek zawarty w definiowanej pętli jest pewnym wyrażeniem, które najczęściej zwraca wartość typu logicznego. Istnieją języki programowania, w których składnia nie przewiduje takiego typu danych.

W językach tych stosuje się wyrażenia zwracające pewną wartość innego typu, która następnie podlega odpowiedniej interpretacji, np. wartość zero może być utożsamiana z wartością **false** typu logicznego, a pozostałe wartości z wartością **true**.

Zapis wyrażenia, oraz typ wartości wyrażenia, zależny jest od składni konkretnego języka programowania. Powszechnym jest zapis wyrażeń kontrolnych analogicznie do zapisu tych wyrażeń dla instrukcji warunkowej.

Szczególne znaczenie mają w tym przypadku operatory porównań, choć warunek może zostać wyrażony także całkiem inaczej, np. jako wywołanie funkcji zwracającej wartość logiczną, bądź jako identyfikator zmiennej logicznej, której wcześniej przypisano rezultat ewaluacji wyrażenia warunkowego.

Ponadto operator logiczny realizujący operację negacji pozwana na rekompensatę ewentualnego braku pętli powtarzanej przy spełnieniu lub niespełnieniu warunku, gdyż zastosowanie tego operatora do podanego warunku jest równoważne zastosowaniu frazy przeciwnej.

Warunki decydujące o kontynuacji lub zaprzestaniu wykonywania pętli mogą być sprawdzane:

- na początku pętli, przed wykonaniem pierwszej instrukcji zawartej w bloku definiowanej pętli,
- wewnątrz pętli, w jej bloku, po wykonaniu części instrukcji,
- na końcu pętli, po wykonaniu wszystkich instrukcji zawartych w bloku definiowanej pętli.

Jeżeli warunek jest sprawdzany na początku pętli, to może nastąpić taka sytuacja, że instrukcje zawarte w pętli nigdy nie zostaną wykonane. Będzie to miało miejsce w sytuacji, gdy przy pierwszym wykonaniu warunek nie będzie spełniony.

Inaczej jest, gdy warunek jest sprawdzany na końcu pętli. W tym przypadku instrukcje zawarte w pętli zostaną wykonane co najmniej jeden raz.

Często pożądane jest, aby instrukcje pętli zostały wykonane dla każdego elementu tablicy, kolekcji itp.

Oczywiście można to zrobić za pomocą pętli iteracyjnych lub repetycyjnych, ale często szybszym i bardziej przejrzystym sposobem jest użycie pętli typu **foreach**, która zwalnia programistę z obowiązku *ręcznego* iterowania po kolekcji.

Działanie pętli foreach, polega na powtarzaniu kolejnych iteracji dla wszystkich elementów wybranego kontenera danych, takiego jak, np. tablica, lista, kolekcja, kolejka, itp.

Pętla taka automatycznie przed przejściem do wykonania kolejnej iteracji przypisuje zadanej w nagłówku pętli zmiennej sterującej wartość kolejnego elementu.

Działanie tej pętli polega na wykonaniu następujących kroków:

- ustaleni jako bieżący, pierwszego element kontenera danych, jeżeli kontener jest pusty, zakończenie działanie pętli,
- przypisanie wartość bieżącego elementu do zmiennej sterującej,
- wykonanie iteracji,
- sprawdzenie czy istnieje kolejny element w kontenerze:
  - jeżeli istnieje, to ustala jako bieżący kolejny element kontenera i wykonuje iterację,
  - jeżeli nie istnieje to kończy działanie.

```
index = 20
counter = 0

while(index > 10):

  index -= 1
  counter += 1
  print("Indeks = %d, licznik = %d" %(index, counter))

→ index = 10, counter = 10
```

Instrukcja stanowiąca ciało iteracji może **nie wykonać się ani razu**.

Wyrażenie występujące w nawiasach określa **warunek kontynuacji**.

Pętla wykonuje się dopóty, dopóki **warunek jest prawdziwy**.

```
counter = 0

for index in range(20, 10, -1):
  counter += 1
  print("Indeks = %d, licznik = %d" %(index, counter))

→ index = 11, counter = 10
```

Pętla `for` w języku Python jest podobna do pętli `foreach` znanych z innych języków.

Pętla jest uniwersalnym **iteratorem** – może przechodzić przez kolejne elementy w dowolnej uporządkowanej sekwencji lub innym obiekcie iterowalnym.

Instrukcja `for` działa na łańcuchach znaków, listach, krotkach, innych wbudowanych obiektach, po których można iterować.

```
L = [2, 4, 6, 8]

for x in L:
  print(x, end = " ")
→ 2 4 6 8
```

```
S = "python"

for x in S:
  print(x, end = " ")
→ p y t h o n
```

```
T = ("ala", "ma", "psa")

for x in T:
  print(x, end = " ")
→ ala ma psa
```

Instrukcje break i continue

Instrukcja `break` pozwala na **natychmiastowe zakończenie** nawet zagnieżdżonej, dowolnej instrukcji **pętli** lub **wyboru**.

Instrukcja `continue` pozwala na **przerwanie bieżącego przebiegu** dowolnej iteracji i przejście do wykonania następnego jej przebiegu.

```
print("Wpisz dowolna literę, zero kończy iterację")

while True:

  key = input("Klawisz: ")

  if(key == '0'):
     break

  if(key >= '0' and key <= '9'):
    print("Wpisałś cyfrę %c o kodzie %d" % (key, ord(key)))
    continue

  print("Wpisałś literę %c o kodzie %d" % (key, ord(key)))
```

Podprogramy

Podprogram – termin związany jest z programowaniem proceduralnym. Podprogram to wydzielona część programu wykonująca jakieś operacje.

Podprogramy stosuje się, aby uprościć program główny, zwiększyć czytelność kodu. Wartościową cechą podprogramu jest możliwość wielokrotnego jego wywołania.

W wielu językach programowania dzieli się podprogramy na funkcje i procedury.

Funkcja ma wykonywać obliczenia i zwracać jakąś wartość, nie powinna natomiast mieć żadnego innego wpływu na działanie programu.

Procedura natomiast nie zwraca żadnej wartości, zamiast tego wykonuje pewne działania.

Przez **zwracanie wartości** należy rozumieć możliwość użycia wywołania funkcji wewnątrz wyrażenia.

Oprócz powyższego podziału, wyróżnić także należy **podprogram główny**, czyli taki od którego rozpoczyna się wykonywanie programu.

Podprogram może być identyfikowany:

- przez **nazwę** – identyfikator przypisany do podprogramu w jego deklaracji, jest to najczęściej spotykany przypadek w językach wysokiego poziomu,
- przez **etykietę**,
- przez **liczbę** – literał całkowity,
- przez **referencję** (adres) – w językach wysokiego poziomu takie wskaźniki przechowywane są w zmiennych typu **proceduralnego** lub **wskaźnikowego**.

Wywołanie podprogramu może być:

- **funkcyjne** – w wyrażeniu, do którego podprogram zwraca obliczoną wartość, taka forma wywołania dotyczy tylko podprogramów mających cechy funkcji, tzn. zwracających wartość,
- **proceduralne** – czyli jako instrukcja, poprzez nazwę z listą argumentów, lub po słowie kluczowym.

Konkretne implementacje języków często dopuszczają wywołanie funkcji w postaci proceduralnej, tzn. poza wyrażeniami. W tym przypadku zwracana przez podprogram wartość jest ignorowana.

Podprogram jako samodzielna, wydzielona część algorytmu, zazwyczaj musi komunikować się z otoczeniem. Taką komunikację realizuje się za pomocą:

- zmiennych globalnych,
- argumentów (parametrów aktualnych), przypisywanych zdefiniowanym w podprogramie,
- rezultatów funkcji (wartości zwracanych do miejsca wywołania),
- pól obiektu (dla metod danego obiektu),
- innych, rzadko stosowanych lub nie zalecanych (np.: obszarów wspólnych, zmiennych nakładanych).

Funkcje w Pythonie są definiowane za pomocą nowej instrukcji `def`.
W przeciwieństwie do funkcji z języków kompilowanych (np. C) `def` jest **instrukcją wykonywalną** – funkcja w Pythonie nie istnieje, dopóki Python nie dotrze do jej kodu i nie wykona instrukcji `def`.

Poprawne (i czasami użyteczne) jest zagnieżdżanie instrukcji `def` wewnątrz innych instrukcji (`if`, `while`), a nawet innych instrukcji `def`.

Kiedy interpreter dotrze do instrukcji `def` i ją wykona, **generowany jest nowy obiekt funkcji**, który zostaje przypisany do **nazwy funkcji**.

Tak jak w przypadku wszystkich operacji przypisania, nazwa funkcji staje się **referencją do obiektu funkcji**. Do obiektów funkcji można także dołączać dowolne zdefiniowane przez użytkownika atrybuty w celu przechowywania danych.

Definicja funkcji

```
def area(a, b):
  return a * b
```

Przykłady wywołania funkcji

```
print(area(2, 3))
```

```
x = area(2, 3)
```

```
x = 2
y = 3

z = area(x, y)
```

Przekazywanie argumentów odbywa się **zawsze przez wartość**, przy czym ta wartość jest **zawsze referencją** do obiektu, a nie wartością tego obiektu.

W praktyce oznacza to, że w przypadku **niemutowalnych** typów danych (np. liczby całkowite, łańcuchy znaków) funkcje w Pythonie działają tak, jakby ich argumenty przekazywane były **przez wartość**.

Przypisanie do nazw argumentów wewnątrz funkcji nie wpływa na wywołującego. Nazwy argumentów w nagłówku funkcji stają się w czasie wykonywania funkcji nowymi zmiennymi lokalnymi w zasięgu funkcji. Nazwy argumentów funkcji i nazwy zmiennych w zasięgu wywołującego nie są aliasami.

Natomiast w przypadku **mutowalnych** typów danych (np. listy, słowniki) na ogół jak **przez referencję**.

Modyfikacja mutowalnego argumentu w funkcji może mieć wpływ na wywołującego. Argumenty są po prostu przypisywane do przekazywanych obiektów, funkcje mogą modyfikować w miejscu przekazywane obiekty typu zmiennego, a wynik może wpłynąć na wywołującego.

Argumenty zmienne mogą być danymi wejściowymi oraz wyjściowymi funkcji.

```
def inc(num):
  num + = 1
  print("Wartość w funkcji:", num)
  return num

num = 10

print("Wartość przed funkcją:", num)

inc(num)

print("Wartość po funkcji:", num)

→ Wartość przed funkcją: 10
→ Wartość w funkcji: 11
→ Wartość po funkcji: 10
```

```python
def change(lst):
  lst.append([1, 2, 3])
  print("Wartość w funkcji: ", lst)
  return lst

lst = ["ala", "ma", "psa"]

print("Wartość przed funkcją:", lst)

change(lst)

print("Wartość po funkcji:", lst)
```

```
→ Wartość przed funkcją: ['ala', 'ma', 'psa']
→ Wartość w funkcji: ['ala', 'ma', 'psa', [1, 2, 3]]
→ Wartość po funkcji: ['ala', 'ma', 'psa', [1, 2, 3]]
```

Makra

Makra (makropolecenia) są prostymi programami, które przechowują serię poleceń zarejestrowanych przez użytkownika za pomocą rejestratora makr wbudowanego w program lub zaprogramowanych ręcznie.

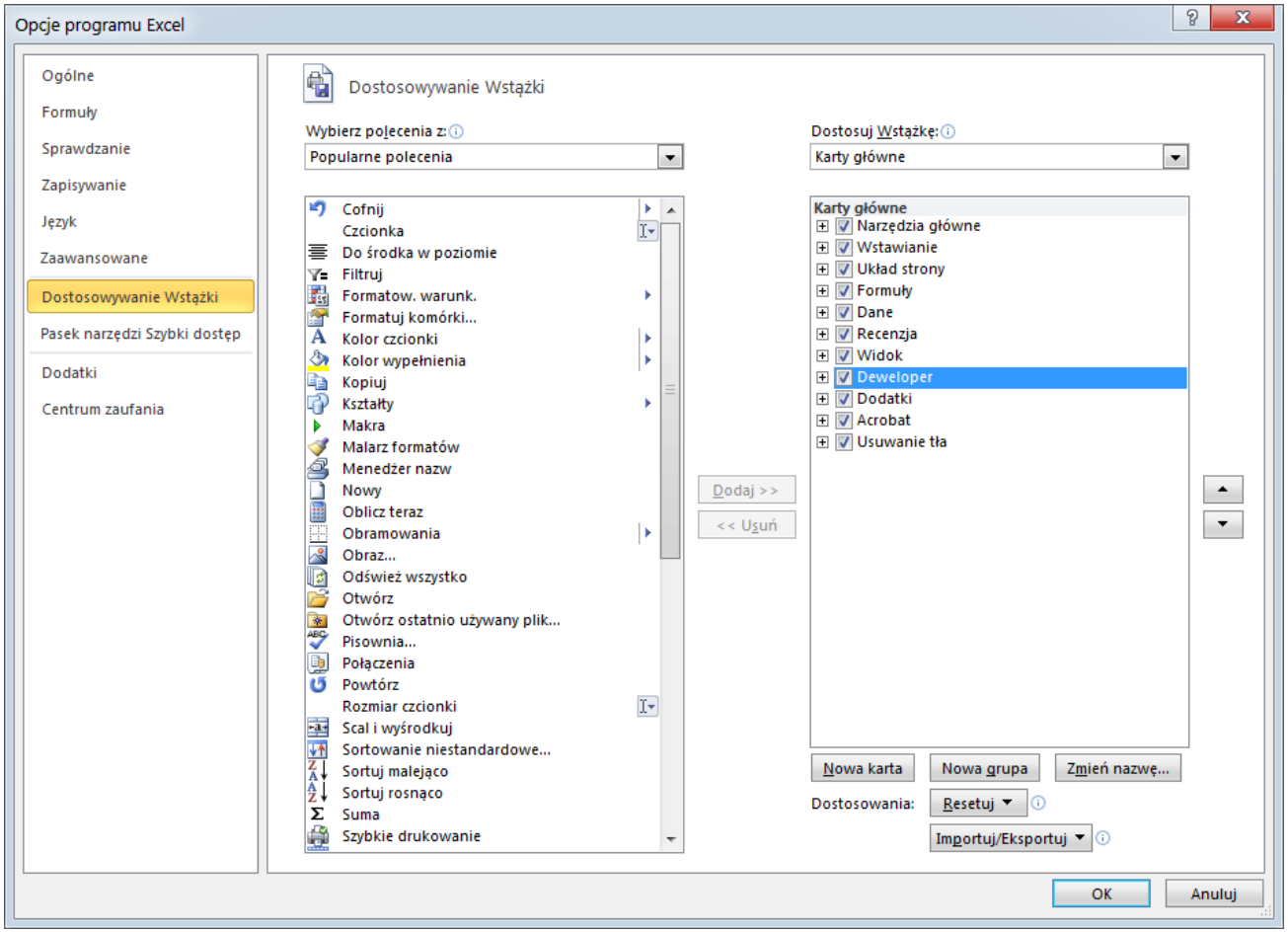
Makra pozwalają na odtworzenie czynności, automatyzację pewnych czynności lub dokonania zmian w dokumentach bez interakcji z użytkownikiem.

Makra pisane są zwykle w skryptowych językach programowania wykonywanych przez interpreter wbudowany w aplikacje, w których są uruchamiane.

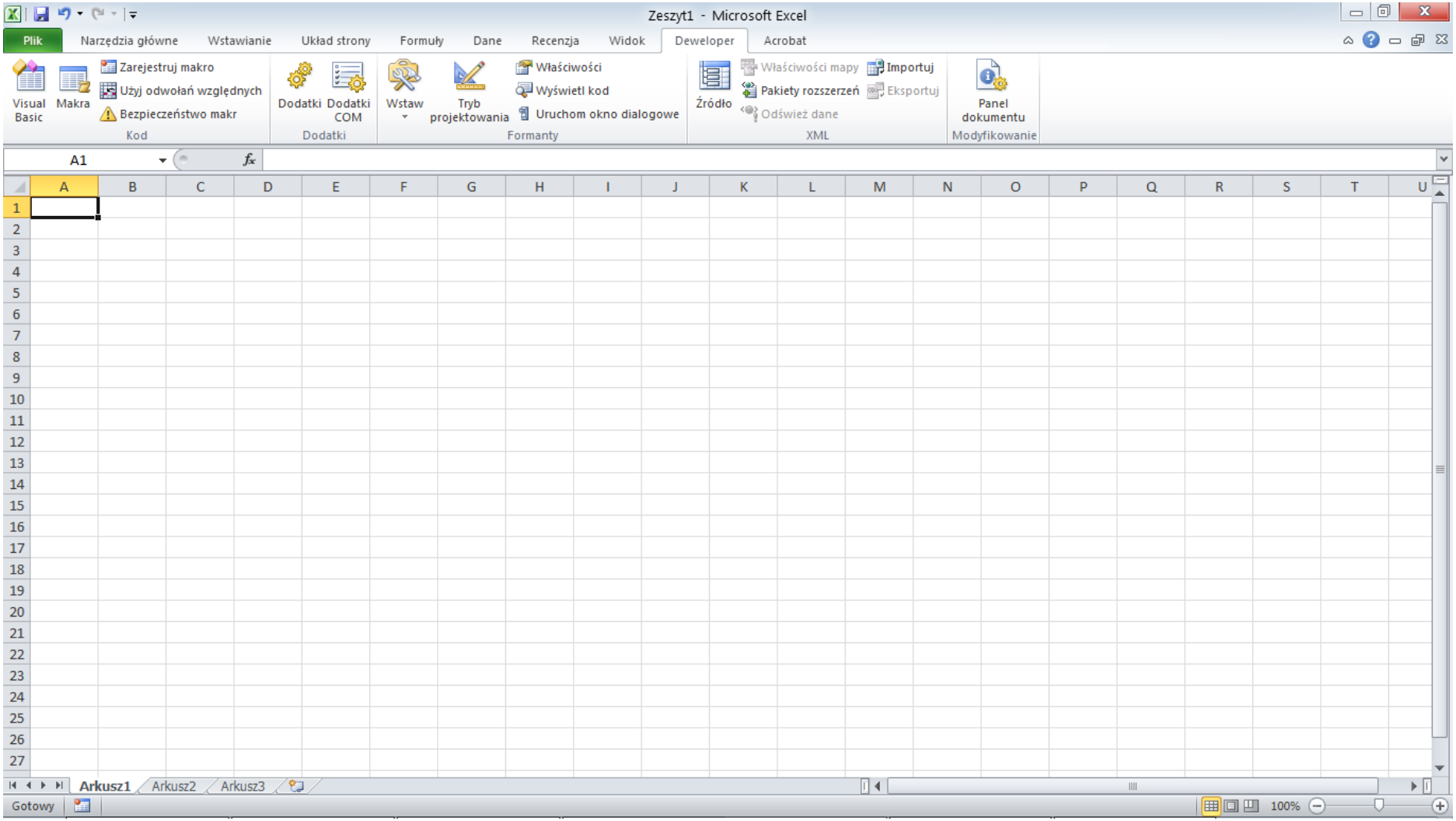
Makra warto tworzyć gdy w programie wykonujemy te same powtarzające się czynności lub gdy program nie oferuje narzędzia do wykonania zamierzonego zadania.

W przypadku Excela makra tworzone są w języku **Visual Basic for Applications** (VBA), który jest wbudowany we wszystkie aplikacje pakietu Microsoft Office.

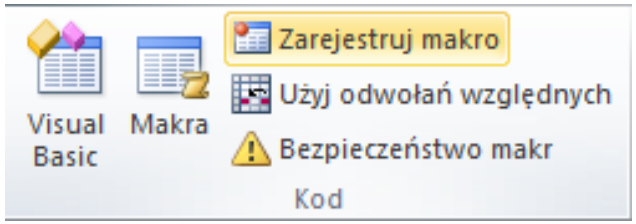
Narzędzia do programowania w VBA (karta **Deweloper**) nie są domyślnie wyświetlane, aby je włączyć należy zaznaczyć pole wyboru **Deweloper** w obszarze **Karty główne** po wybraniu opcji: **Plik | Opcje | Dostosowywanie Wstążki**.

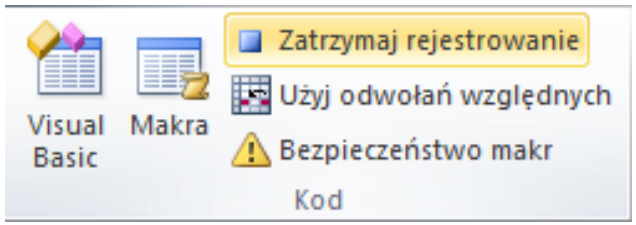


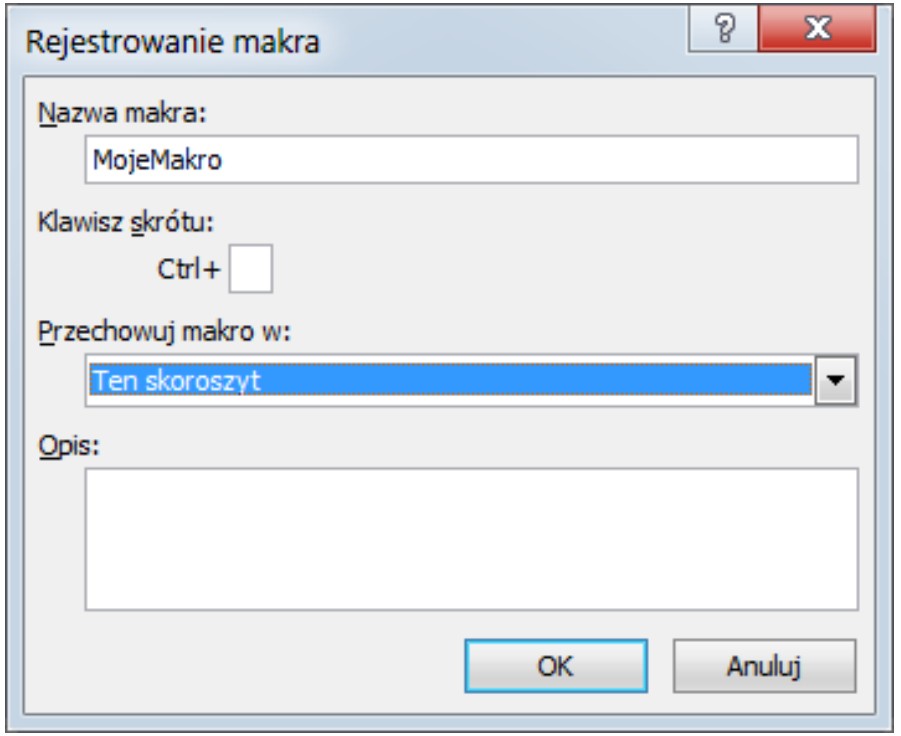
Wstążka **Deweloper**.



Rejestrowanie makr







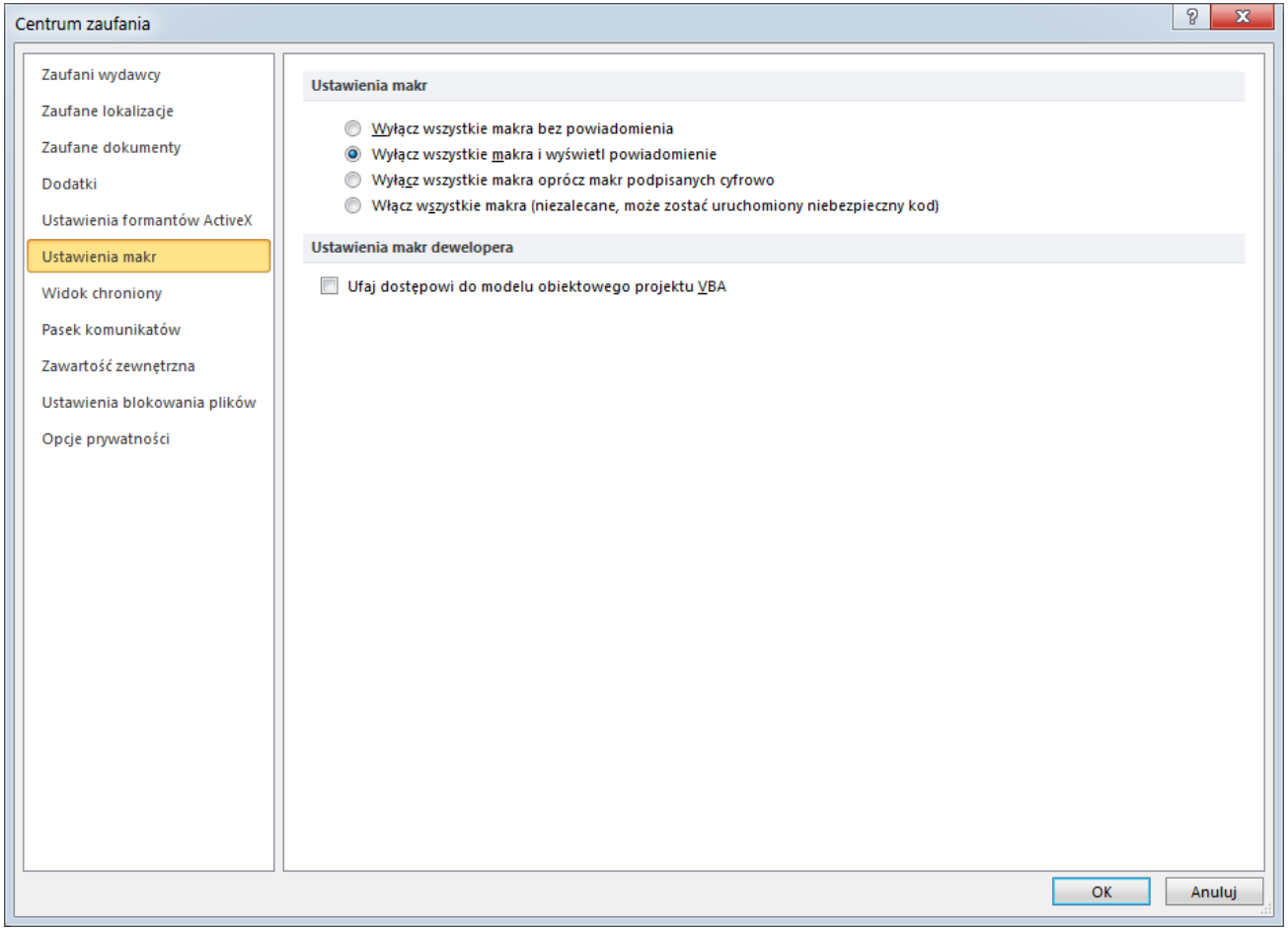
Odwołania:

- odwołanie względne,
- odwołanie bezwzględne.

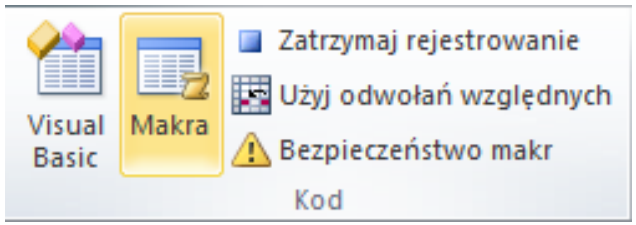
Miejsce przechowania makra:

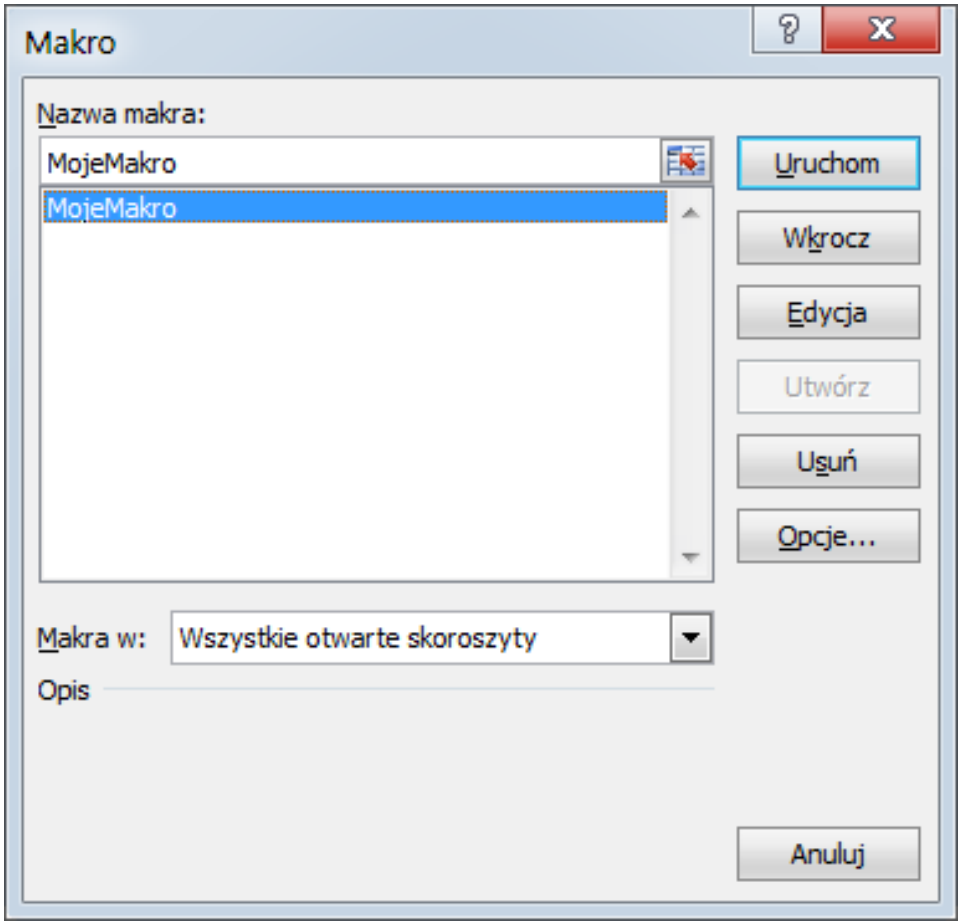
- skoroszyt makr osobistych (`PERSONAL.XLSB`),
- nowy skoroszyt,
- ten skoroszyt.

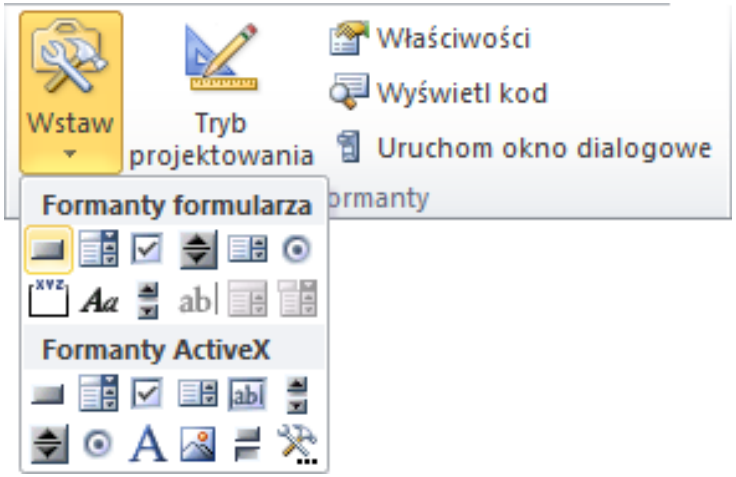
Poziom bezpieczeństwa

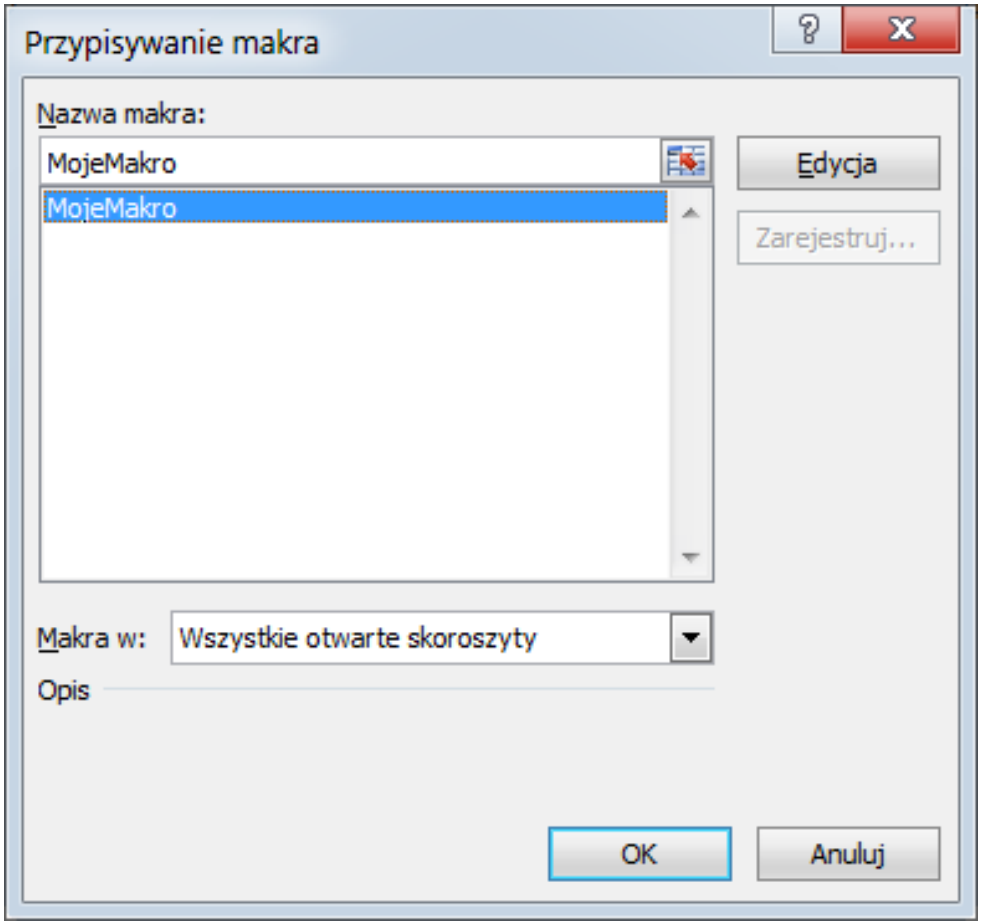


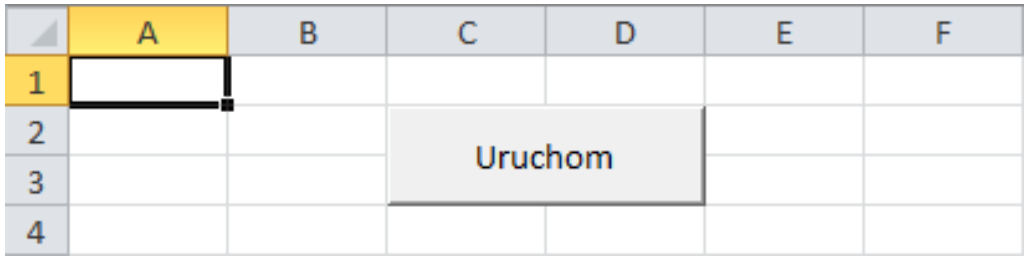
Uruchomienie makra



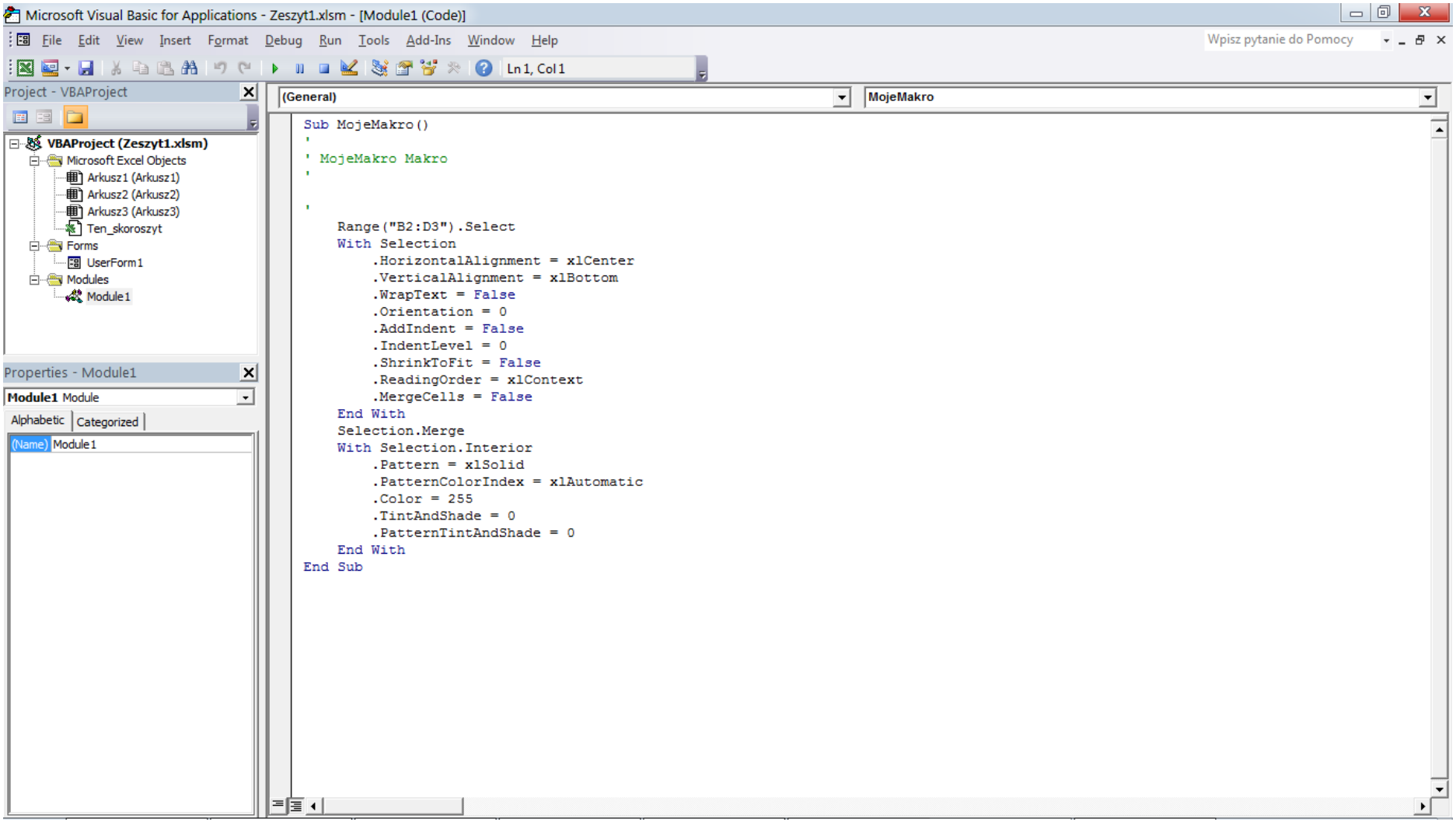


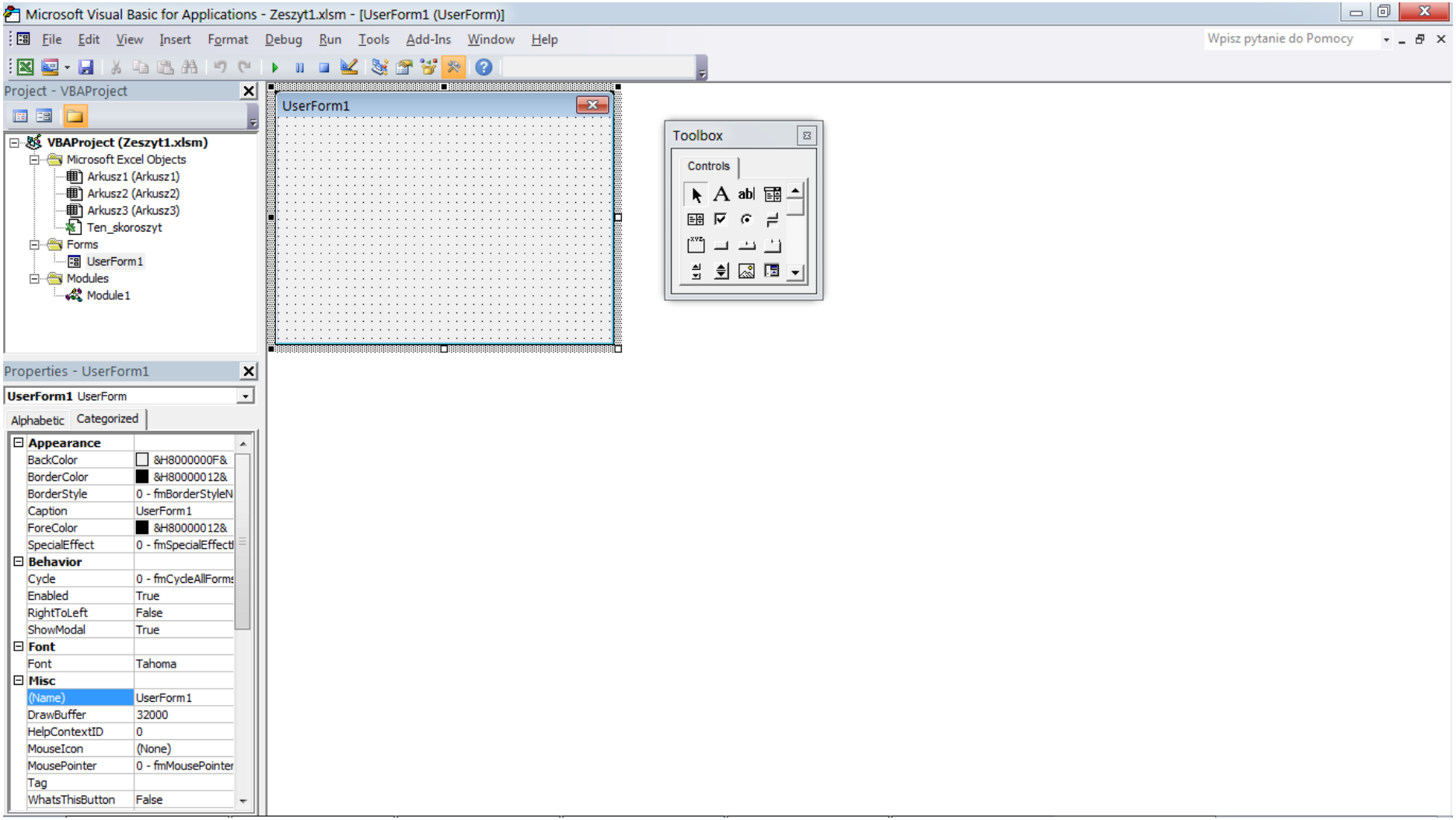

Wstaw
Tryb projektowania
Właściwości
Wyświetl kod
Uruchom okno dialogowe
Formanty formularza
Formanty ActiveX

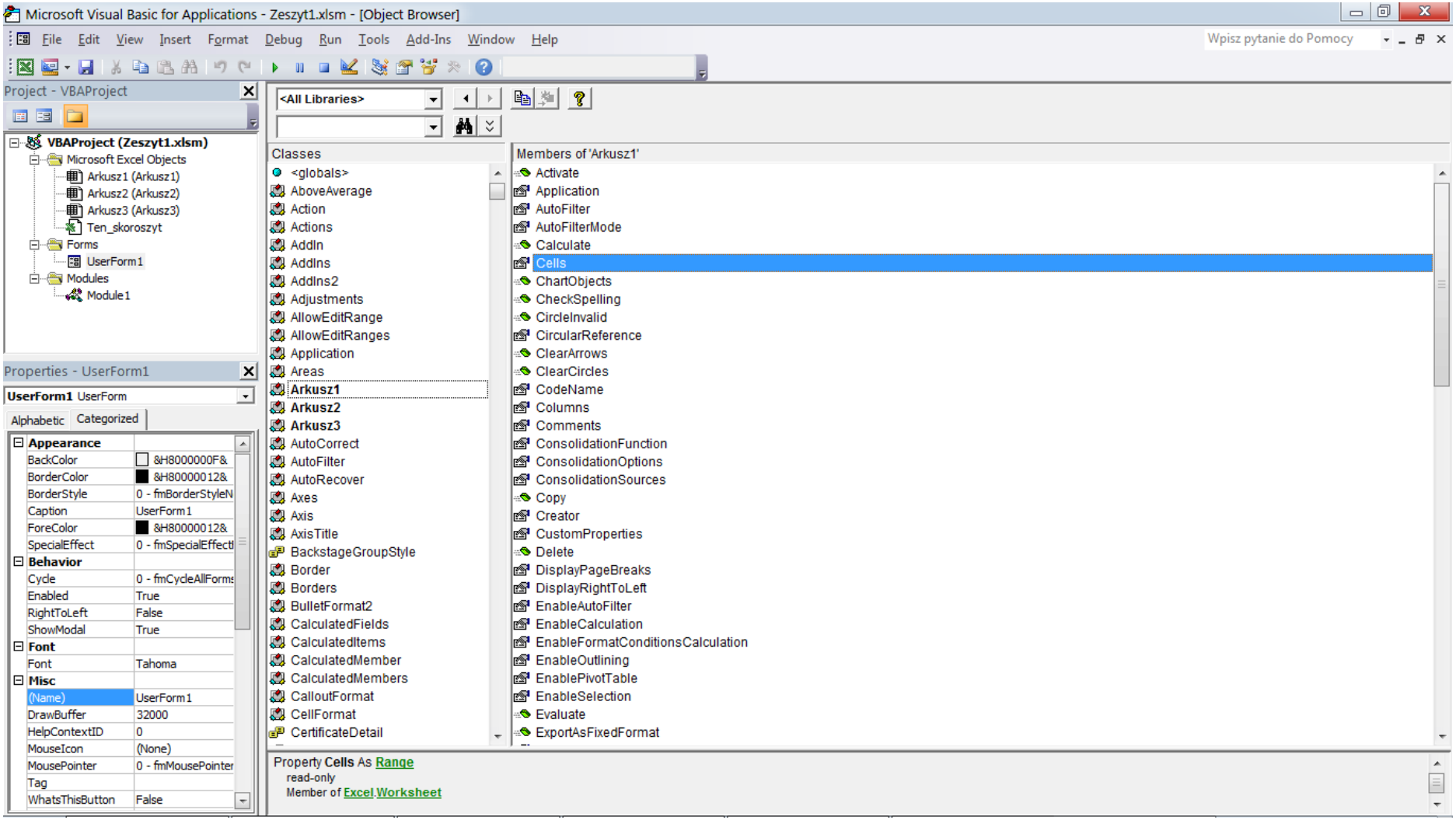

Przypisywanie makra
Nazwa makra:
MojeMakro
MojeMakro
Edycja
Zarejestruj...
Makra w:
Wszystkie otwarte skoroszyty
Opis
OK
Anuluj


A
B
C
D
E
F
1
2
3
4
Uruchom

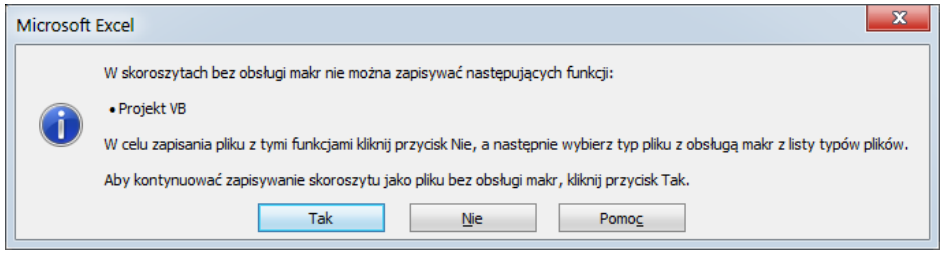
Edytor Visual Basic

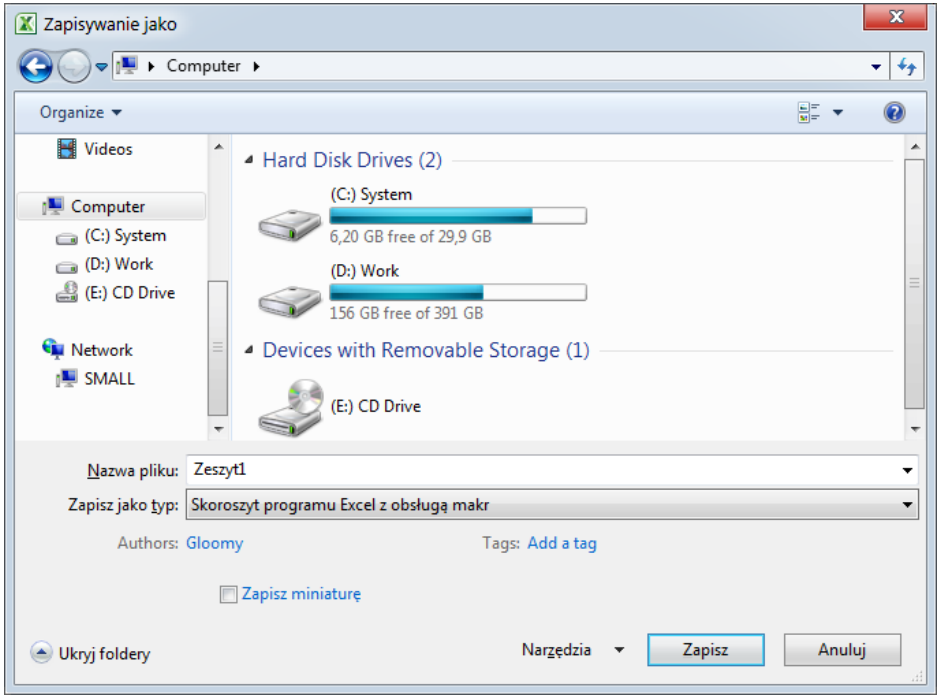



Microsoft Visual Basic for Applications - Zeszyt1.xlsm - [UserForm1 (UserForm)]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Wpisz pytanie do Pomocy
Project - VBAProject
VBAProject (Zeszyt1.xlsm)
Microsoft Excel Objects
Arkusz1 (Arkusz1)
Arkusz2 (Arkusz2)
Arkusz3 (Arkusz3)
Ten_skoroszyt
Forms
UserForm1
Modules
Module1
UserForm1
Toolbox
Controls
Properties - UserForm1
UserForm1 UserForm
Alphabetic Categorized
Appearance
BackColor &H8000000F&
BorderColor &H80000012&
BorderStyle 0 - fmBorderStyleN
Caption UserForm1
ForeColor &H80000012&
SpecialEffect 0 - fmSpecialEffect
Behavior
Cycle 0 - fmCycleAllForms
Enabled True
RightToLeft False
ShowModal True
Font
Font Tahoma
Misc
(Name) UserForm1
DrawBuffer 32000
HelpContextID 0
MouseIcon (None)
MousePointer 0 - fmMousePointer
Tag
WhatsThisButton False

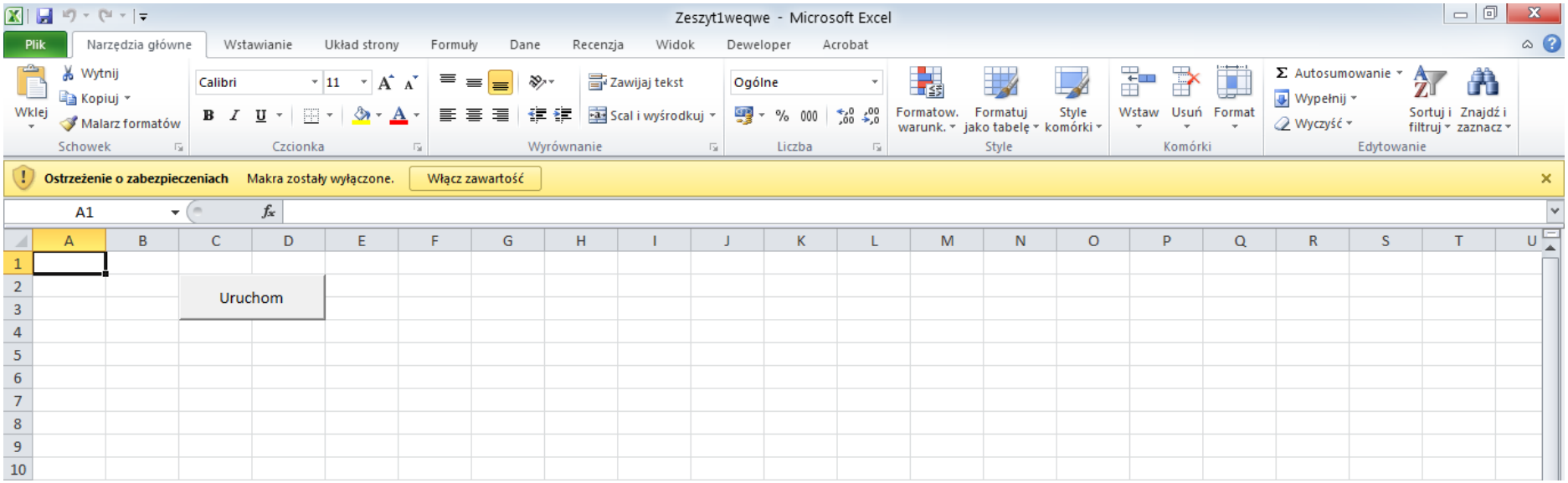

Microsoft Visual Basic for Applications - Zeszyt1.xlsm - [Object Browser]
File Edit View Insert Format Debug Run Tools Add-Ins Window Help
Wpisz pytanie do Pomocy
Project - VBAProject
VBAProject (Zeszyt1.xlsm)
Microsoft Excel Objects
Arkusz1 (Arkusz1)
Arkusz2 (Arkusz2)
Arkusz3 (Arkusz3)
Ten_skoroszyt
Forms
UserForm1
Modules
Module1
Properties - UserForm1
UserForm1 UserForm
Alphabetic Categorized
Appearance
BackColor &H8000000F&
BorderColor &H80000012&
BorderStyle 0 - fmBorderStyleN
Caption UserForm1
ForeColor &H80000012&
SpecialEffect 0 - fmSpecialEffect
Behavior
Cycle 0 - fmCycleAllForms
Enabled True
RightToLeft False
ShowModal True
Font
Font Tahoma
Misc
(Name) UserForm1
DrawBuffer 32000
HelpContextID 0
MouseIcon (None)
MousePointer 0 - fmMousePointer
Tag
WhatsThisButton False
<All Libraries>
Classes
<globals>
AboveAverage
Action
Actions
AddIn
AddIns
AddIns2
Adjustments
AllowEditRange
AllowEditRanges
Application
Areas
Arkusz1
Arkusz2
Arkusz3
AutoCorrect
AutoFilter
AutoRecover
Axes
Axis
AxisTitle
BackstageGroupStyle
Border
Borders
BulletFormat2
CalculatedFields
CalculatedItems
CalculatedMember
CalculatedMembers
CalloutFormat
CellFormat
CertificateDetail
Members of 'Arkusz1'
Activate
Application
AutoFilter
AutoFilterMode
Calculate
Cells
ChartObjects
CheckSpelling
CircleInvalid
CircularReference
ClearArrows
ClearCircles
CodeName
Columns
Comments
ConsolidationFunction
ConsolidationOptions
ConsolidationSources
Copy
Creator
CustomProperties
Delete
DisplayPageBreaks
DisplayRightToLeft
EnableAutoFilter
EnableCalculation
EnableFormatConditionsCalculation
EnableOutlining
EnablePivotTable
EnableSelection
Evaluate
ExportAsFixedFormat
Property Cells As Range
read-only
Member of Excel.Worksheet

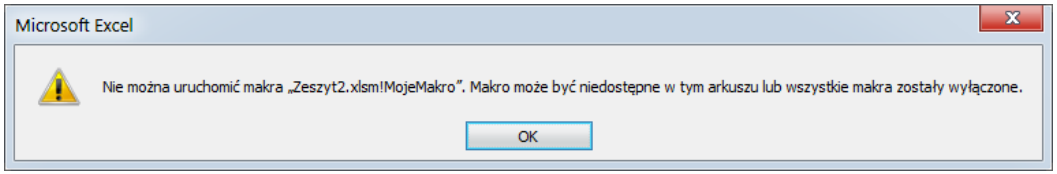
Zapisywanie skoroszytów z makrami





Otwieranie skoroszytów z makrami





Obiekty, właściwości, metody

W przypadku Excela, przez **obiekt** będziemy rozumieć *zasób* kontrolowany przez język Visual Basic. Skoroszyt, arkusz, zakres komórek, wykres to tylko niektóre przykłady.

Sam Excel stanowi obiekt – aplikację. Zawiera też inne obiekty takie jak skoroszyt czy paski narzędzi. Skoroszyt z kolei zawiera kolejne obiekty takie jak arkusze, wykresy.

Takie same lub podobne obiekty stanowią tzw. **kolekcje**. Kolekcja arkuszy obejmuje wszystkie arkusze w skoroszycie. Najczęściej używanymi kolekcjami są: `Worksheets`, `Sheets` (zawierająca arkusze i wykresy), `Workbooks`, `Windows`.

Pracując z kolekcją, obiektów możemy wykonać te same czynności dla wszystkich obiektów w kolekcji.

Każdy obiekt ma pewne cechy charakterystyczne, które noszą nazwę **właściwości**. Właściwości obiektów można ustawiać lub odczytywać.

Na przykład obiekt `Workbook` ma właściwość `Name`, natomiast obiekt `Range` ma właściwości: `Column`, `Font`, `Formula`, `Name`, `Row`, `Style`, `Value`.

W Visual Basic, niektóre właściwości mogą być również obiektami. Zakres komórek `Range` jest właściwością obiektu `Worksheet`, ale możemy zmieniać wygląd czcionki zakresu ustawiając właściwość `Font`, która z kolei ma inne właściwości (z tego wynika, że `Range` i `Font` są obiektami).

Oprócz właściwości, obiekty posiadają **metody**. Metoda jest to czynność, którą może wykonać dany obiekt (metoda jest wykonywana na obiekcie).

Na przykład, obiekt `Range` ma metody `ClearContents` i `ClearFormats`.

Metody mogą mieć opcjonalne parametry, które wpływają na ich zachowanie. Obiekt `Workbook` ma metodę `Close`, która zamyka otwarty skoroszyt.

Jeśli w skoroszycie są niezapisane zmiany, Excel wyświetli komunikat z pytaniem, czy je zachować na dysku.

Aby zamknąć skoroszyt bez zachowania zmian, należy metodę `Close` uruchomić z parametrem `SaveChanges` ustawionym na `False`.

```
Range
Worksheet.Range
Workbook.Worksheet.Range
Applcation.Workbook.Worksheet.Range
```

```
Range("A1").ClearContents
```

```
Workbooks("Zeszyt1.xlsm").Worksheets("Arkusz1").Range("A1").ClearContents
```

Syntaktyka i semantyka VBA

Odwołanie się do właściwości obiektu:

Obiekt.Wlasciwosc

```
Range("A1").Value
```

Obiekt.Wlasciwosc(Argumenty)

```
ActiveCell.Offset(columnOffset:=2, rowOffset:=3)
```

Zmiana wartości właściwości:

Obiekt.Wlasciwosc = Wartosc

```
Range("A1").Value = 25
```

```
Range("A1").Formula = "=1+2*3"
```

```
ActiveCell.Font.Bold = True
```

```
ActiveCell.Font.Name = "Times New Roman"
```

Odczytanie wartości właściwości:

Zmienna = Obiekt.Wlasciwosc

```
Variable = Range("A1").Value
```

Odwołanie do metody obiektu:

Obiekt.Metoda

```
Range("A1").ClearContents
```

Podział długich linii:

```
Selection.PasteSpecial _
  Paste:=xlValues, _
  Operation:=xlMultiply, _
  SkipBlanks:=False, _
  Transpose:=False
```

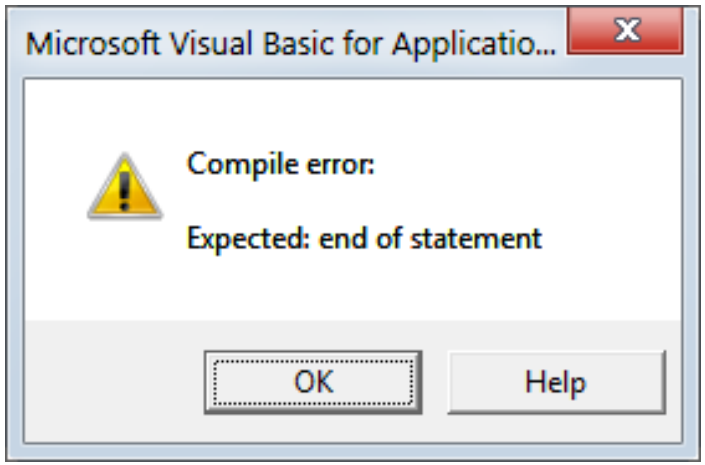
Instrukcja może zawierać 1024 znaki. Użycie znaku kontynuacji wiersza jest dozwolone:

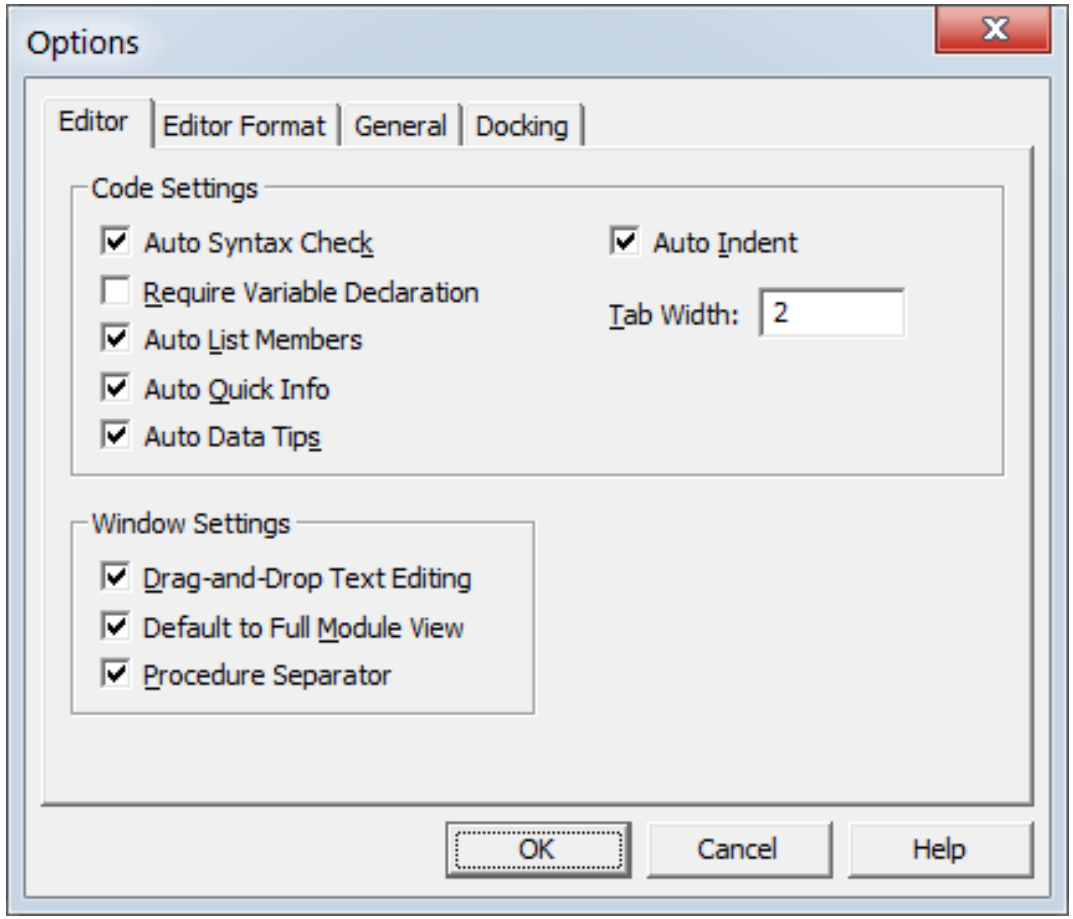
- przed lub po operatorach `&`, `+`, `Like`, `NOT`, `AND`,
- przed przecinkiem lub po przecinku,
- przed lub po przypisaniu (`:=`),
- przed lub po znaku równości (`=`).

Błędy

W trakcie pisania kodu nie da się uniknąć błędów. Błędy mogą wynikać z niepoprawnego wpisania instrukcji, pominięcia kropek, przecinków, nawiasów, etc. Takie błędy noszą nazwę **syntaktycznych** (składniowych).

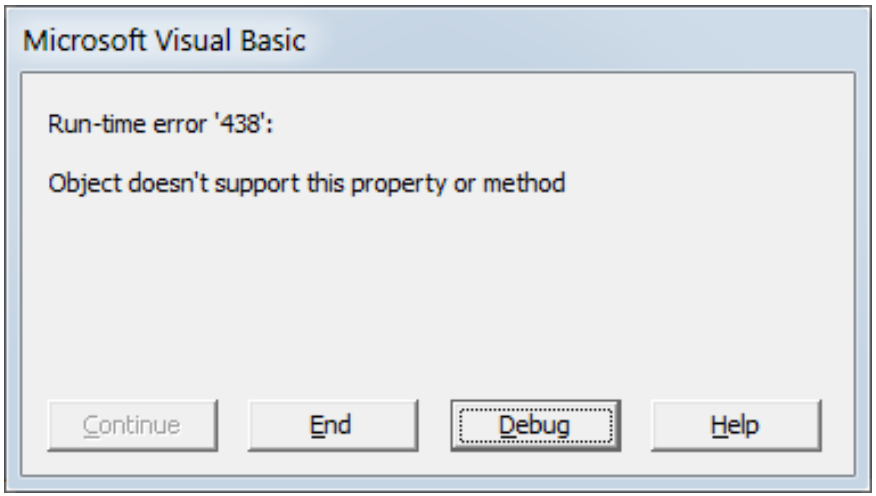
Edytor Visual Basic automatycznie sprawdza błędy takiego typu w trakcie pisania kodu.

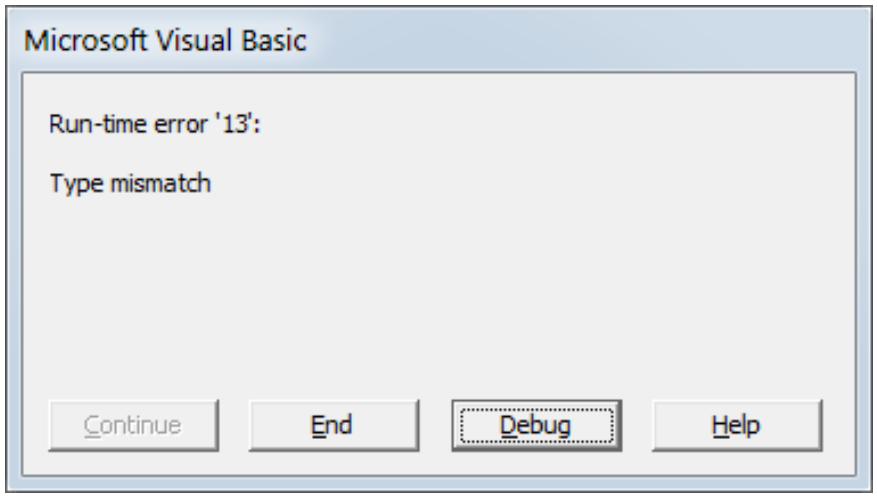

Microsoft Visual Basic for Applicatio...
Compile error:
Expected: end of statement
OK
Help


Options
Editor
Editor Format
General
Docking
Code Settings
Auto Syntax Check
Require Variable Declaration
Auto List Members
Auto Quick Info
Auto Data Tips
Auto Indent
Tab Width: 2
Window Settings
Drag-and-Drop Text Editing
Default to Full Module View
Procedure Separator
OK
Cancel
Help

Oprócz błędów syntaktycznych, można spotkać jeszcze błędy **semantyczne** (znaczeniowe, wykonania) i **logiczne**.

Błędy wykonania występują w chwili odtwarzania procedur (programu) i mogą wynikać z próby uruchomienia nieistniejącej procedury, otwarcia nieistniejącego pliku lub złego typowania zmiennych.

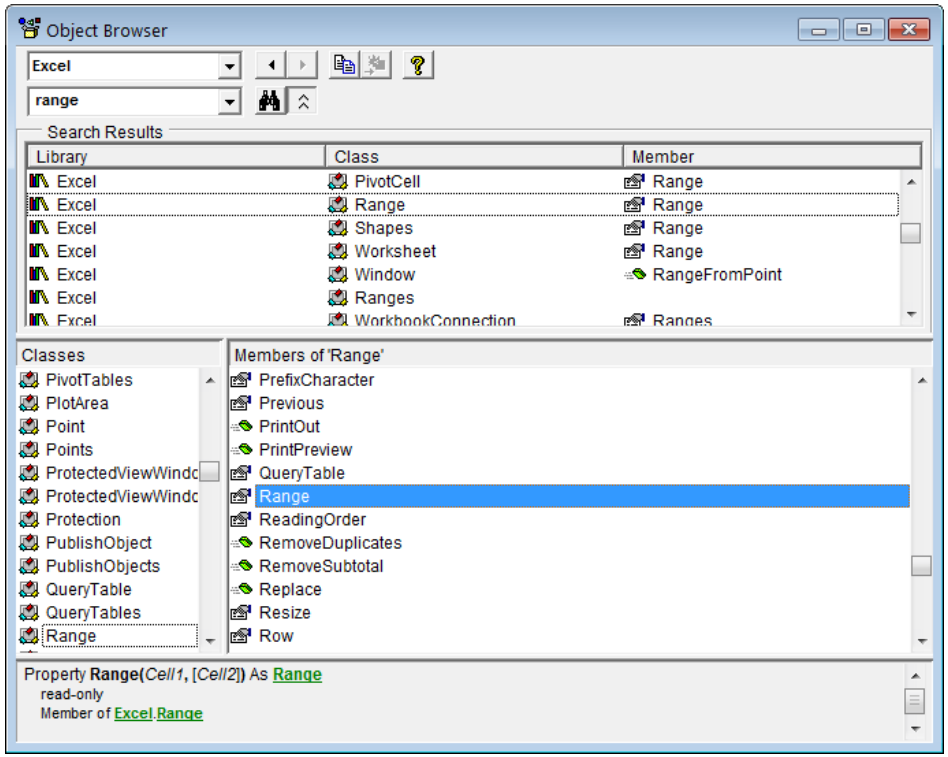




Podpowiedzi i przeglądarka obiektów

```
Range (
```
Range(***Cell1***, [*Cell2*]) As Range

```
Range("A1").
```
- Activate
- AddComment
- AddIndent
- Address
- AddressLocal
- AdvancedFilter
- AllocateChanges



Operacje związane z komórkami

właściwość `Range`	
wybranie pojedynczej komórki	`Range("A1").Select`
wybranie zakresu komórek	`Range("A1:C5").Select`
wybranie kilku komórek	`Range("A1, B2, C3").Select`
wybranie kilku komórek i zakresu	`Range("A1:C5, E7, F9").Select`

właściwość `Cells`	
wybranie pojedynczej komórki	`Cells(5, 1).Select`
	`Cells(5, "A").Select`
wybranie zakresu komórek	`Range(Cells(5, 1), Cells(10, 2)).Select`
wybranie wszystkich komórek	`Cells.Select`

właściwość `Offset`	
wybranie komórki znajdującej się wiersz niżej i trzy kolumny w prawo w stosunku do A1	`Range("A1").Offset(1, 3).Select`
wybranie komórki znajdującej się wiersz wyżej i jedną kolumnę w lewo w stosunku do E5	`Range("E5").Offset(-2, -1).Select`

komórki skrajne	
pierwsza komórka w wierszu	`ActiveCell.End(xlleft).Select`
ostatnia komórka w wierszu	`ActiveCell.End(xlright).Select`
pierwsza komórka w kolumnie	`ActiveCell.End(xlup).Select`
ostatnia komórka w kolumnie	`ActiveCell.End(xldown).Select`

przesuwanie, kopiowanie, usuwanie	
przesuwanie zawartości z komórki A1 do A5	`Range("A1").Cut Destination:=Range("A5")`
kopiowanie zawartości z komórki A1 do A5	`Range("A1").Copy Destination:=Range("A5")`
usuwanie zawartości z komórki A1	`Range("A1").Clear`

Typy zmiennych w VBA

typ, wielkość, charakterystyka		
`Boolean`	2	wartość logiczna `True` lub `False`
`Byte`	1	liczba całkowita od `0` do `255`
`Integer`	2	liczba całkowita od `-32768` do `32767`
`Long`	4	liczba całkowita długa od `-2147483648` do `2147483647`
`Single`	4	liczba rzeczywista od `-3.4028235e+38` do `-1.401298e-45` od `1.401298e-45` do `3.4028235e+38`
`Double`	8	liczba rzeczywista długa od `-1.79769313486231570e+308` do `-4.94065645841246544e-324` od `4.94065645841246544e-324` do `1.79769313486231570e+308`

typ, wielkość, charakterystyka		
`Currency`	8	liczba całkowita z kropką (skalowana 10000) od `-922,337,203,685,477.5808` do `922,337,203,685,477.5807`
`Decimal`	12	liczba dziesiętna, ułamek dziesiętny (skalowana potęgami 10) od $\pm$`79 228 162 514 264 337 593 543 950 335` do $\pm$`7.9228162514264337593543950335`
`Date`	8	data od `01/01/100` do `31/12/9999`
`String` stałej długości	-	łańcuch znaków od `0` do `65535` (64kB)
`String` zmiennej długości	-	łańcuch znaków od `0` do ok. 2 mld.

typ, wielkość, charakterystyka		
`Object`	4	referencja na obiekt
`Array`	-	tablica
`Variant`	-	może zawierać wszystkie typy za wyjątkiem łańcucha o stałej długości

Deklarowanie zmiennych

VBA nie wymaga deklarowania zmiennych (deklarowanie niejawne, domniemanie typu). Można od razu do nazw zmiennych przypisywać wartości. **Nie jest to zalecane!**

W przypadku niedeklarowania zmiennych, VBA nie będzie znał i sprawdzał nazw zmiennych, tylko utworzy nową zmienną jeśli napotka na nową (być może tylko błędnie wpisaną nazwę używanej już zmiennej) nazwę zmiennej.

Wszystkie zmienne niedeklarowane będą miały typ `Variant` i będą tworzone podczas wykonywania programu, co może wpłynąć na szybkość działania programu. Mogą się też ujawnić błędy związane z niedozwolonymi operacjami dla konkretnych typów.

Można wymusić deklarowanie zmiennych (deklarowanie jawne) poleceniem (**zalecane**):

```
Option Explicit
```

Aby zadeklarować zmienną, należy podać jej nazwę i typ poprzedzonych instrukcją `Dim`:

```
Option Explicit

Dim dataUrodzenia As Date
Dim imieNazwisko As String
Dim kwota As Single
Dim wiek As Integer
```

Można deklarować wiele zmiennych w jednej instrukcji:

```
Option Explicit

Dim dataUrodzenia As Date, imieNazwisko As String
```

Przypisywanie wartości do zmiennych:

```
Option Explicit

Dim dataUrodzenia As Date
Dim imieNazwisko As String
Dim kwota As Single
Dim wiek As Integer

dataUrodzenia = #01/03/1999#
imieNazwisko = "Jan Kowalski"
kwota = 3.14
wiek = 21
```

Do typowania zmiennych można używać znaków

znaki deklaracji typów	
`Integer`	%
`Long`	&
`Single`	!
`Double`	#
`Currency`	@
`String`	$

```
Sub obliczWiek()

  Dim imieNazwisko$, wiek%
  Dim dataUrodzenia As Date

  imieNazwisko = "Jan Kowalski"
  dataUrodzenia = #01/03/1999#
  wiek = Year(Now()) - Year(dataUrodzenia)

  MsgBox Prompt:=imieNazwisko & " ma " & wiek & " lat."
' MsgBox(imieNazwisko & " ma " & wiek & " lat.")

End Sub
```

Zasięg zmiennych

Można deklarować zmienne mające zasięg:

- procedury,
- modułu,
- projektu.

Poziom procedury:

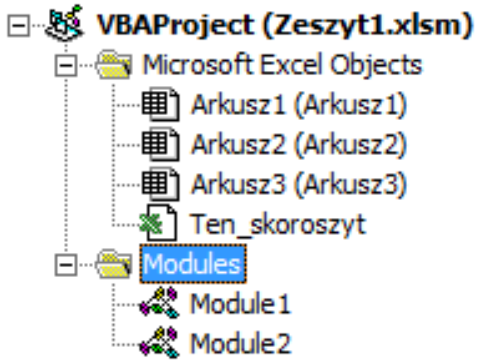
```
Option Explicit

Sub wyswietlLiczbe()

  Dim liczba As Integer

  MsgBox Prompt:="Liczba = " & liczba

End Sub
```

Poziom modułu:

```
Option Explicit

Dim liczba As Integer

Sub wyswietlLiczbe()

  MsgBox Prompt:="Liczba = " & liczba

End Sub

Sub wyswietlLiczbeInaczej()

  MsgBox Prompt:="Liczba wynosi " & liczba

End Sub
```

Poziom projektu:

```
Option Explicit

Public liczba As Integer
```



Zmienna może być używana we wszystkich modułach projektu (`Module1`, `Module2`, itd.).

Zmienne obiektowe, statyczne, stałe

Zmienna statyczna:

```
Option Explicit

Static kosztJednostkowy As Currency
```

Po wyjściu z procedury wartość zmiennej statycznej nie ulega zmianie.

Zmienna obiektowa:

```
Option Explicit

Dim pweienZakres As Object

Set pewienZakres = Worksheet("Arkusz1").Range("A1:C5")
```

Stałe:

```
Option Explicit

Public Const sciezkaDoPliku As String = "c:\files\data.txt"

Const znanaWartosc = 3.14
Const ulubionyKolorIndeks = 4
Const wiekEmerytalny As Integer = 99
```

konwersja typów danych	
`CBool()`	konweercja na typ `Boolean`
`CByte()`	konweercja na typ `Byte`
`CCur()`	konweercja na typ `Currency`
`CDate()`	konweercja na typ `Date`
`CDbl()`	konweercja na typ `Double`
`CDec()`	konweercja na typ `Decimal`
`CInt()`	konweercja na typ `Integer`
`CLng()`	konweercja na typ `Long`
`CSng()`	konweercja na typ `Single`
`CStr()`	konweercja na typ `String`
`CVar()`	konweercja na typ `Variant`

```
Dim intA As Integer
Dim boolA As Boolean

Dim sngB As Single
Dim intB As Integer

Dim dblC As Double
Dim strC As String

intA = 0
boolA = CBool(intA) 'False

sngB = 3.14
intB = CInt(sngB) '3

dblC = 21.12345
strC = CStr(dblC) '"21.12345"
```

sprawdzanie typów danych	
`IsArray()`	czy jest tablicą?
`IsDate()`	czy jest datą?
`IsEmpty()`	czy jest puste?
`IsNumeric()`	czy jest typem numerycznym?
`IsObject()`	czy jest obiektem?

```
Dim tablica1(5) As Integer, tablica2(), wynik As Boolean

tablica2 = Array(1, 2, 3)

wynik = IsArray(tablica1) 'True
wynik = IsArray(tablica2) 'True
```

```vba
Dim zmienna, wynik As Boolean

wynik = IsEmpty(zmienna) 'True

zmienna = Null
wynik = IsEmpty(zmienna) 'False

zmienna = Empty
wynik = IsEmpty(zmienna) 'True
```

```vba
Dim zmienna, wynik As Boolean
Dim wartosc As Integer, zakres As Object, wynik As Boolean

Set zakres = Worksheet("Arkusz1").Range("A1:C5")
wartosc = 3

wynik = IsObject(zakres) 'True
wynik = IsObject(wartosc) 'False
```

Tablice

VBA pozwala na stosowanie jedno- i wielowymiarowych tablic:

- **statycznych**, które posiadają stałą, niezmienną w trakcie działania programu wielkość (liczbę elementów), ustaloną w trakcie deklaracji,
- **dynamicznych**, w których można zmieniać wielkość, w trakcie działania programu, wielkość musi być zadeklarowana przed pierwszym użyciem.

W VBA można zmieniać sposób numerowania elementów tablic.

Deklaracja:

```
Option Base 1
```

spowoduje, że pierwszy element tablicy będzie miał numer 1, natomiast

```
Option Base 0
```

spowoduje, że pierwszy element tablicy będzie miał numer 0.

Deklarowanie i używanie tablic jednowymiarowych:

```
Option Base 1

Dim napis, trzyWioski(3) As String
Dim suma, ulubioneNumerki(5) As Integer

trzyWioski(1) = "Rzepin"
trzyWioski(2) = "Pawłów"
trzyWioski(3) = "Ambrożów"

ulubioneNumerki(1) = 3
ulubioneNumerki(2) = 7
ulubioneNumerki(3) = 9
ulubioneNumerki(4) = 13
ulubioneNumerki(5) = 21

suma = ulubioneNumerki(1) + ulubioneNumerki(3)
napis = trzyWioski(1) & " " & trzyWioski(2)
```

Deklarowanie i używanie tablic wielowymiarowych:

```
Option Base 1

Dim sumaDiag, macierz(3, 3) As Integer

macierz(1, 1) = 1
macierz(1, 2) = 2
macierz(1, 3) = 3
macierz(2, 1) = 4
macierz(2, 2) = 5
macierz(2, 3) = 6
macierz(3, 1) = 7
macierz(3, 2) = 8
macierz(3, 3) = 9

sumaDiag = macierz(1, 1) + macierz(2, 2) + macierz(3, 3)
```

Deklarowanie i używanie tablic dynamicznych:

```
Option Base 1

Dim wektor() As Integer
Dim macierz(,) As Integer

ReDim wektor(3)
ReDim macierz(2, 2)

wektor(1) = 1
wektor(2) = 2
wektor(3) = 3

macierz(1, 1) = 1
macierz(1, 2) = 2
macierz(2, 1) = 3
macierz(2, 2) = 4
```

W VBA można zmieniać wielkość tablic:

```
Option Base 1

Dim wektor(2) As Integer

ReDim wektor(3)
```

Zmiana wielkości tablicy, powoduje utratę jej zawartości. Jeśli chcemy zachować dotychczasowe wartości, trzeba użyć konstrukcji:

```
Option Base 1

Dim ciag(5) As Integer

ciag(1) = 1
ciag(2) = 2
ciag(3) = 3
ciag(4) = 4
ciag(5) = 5

ReDim Preserve ciag(10) 'ciag = {1 2 3 4 5 0 0 0 0 0}
ReDim Preserve ciag(3) 'ciag = {1 2 3}
```

W przypadku tablic wielowymiarowych, można zmieniać tylko ostatni wymiar:

```
Option Base 1

Dim macierz(2, 2) As Integer

macierz(1, 1) = 1
macierz(1, 2) = 2
macierz(2, 1) = 3
macierz(2, 2) = 4

ReDim Preserve macierz(2, 5)

'macierz = {1 2 0 0 0}
'          {3 4 0 0 0}
```

Tablice o nieustalonych rozmiarach można inicjować wartościami za pomocą `Array()`, tablica musi być typu `Variant`, lub będzie mieć typ `Variant`:

```
Option Base 1

Dim cyferki() As Variant
Dim literki() 'deklaracja typu nie jest konieczna

cyferki = Array(1, 2, 3, 4, 5)
literki = Array("a", "b", "c", "d")
```

Badanie rozmiaru tablicy:

```
Option Base 1

Dim tablica(100, 5 To 10, -3 To 3) As Integer
Dim iLow, iUp, jLow, jUp, kLow, kUp As Long

iLow = LBound(tablica, 1) '1
jLow = LBound(tablica, 2) '5
kLow = LBound(tablica, 3) '-3

iUp = UBound(tablica, 1) '100
jUp = UBound(tablica, 2) '10
kUp = UBound(tablica, 3) '3
```

Usuwanie zawartości tablicy:

```
Option Base 1

Dim cyferki(3) As Integer
Dim literki(2, 2) As String

cyferki(1) = 1
cyferki(2) = 2
cyferki(3) = 3

literki(1, 1) = "a"
literki(1, 2) = "b"
literki(2, 1) = "c"
literki(2, 2) = "d"

Erase(cyferki)
Erase(literki)
```

Instrukcja przypisania

W VBA instrukcją przypisania jest znak równości =.

Operator przypisania to operator, który powoduje przypisanie, i zwraca wartość równą wartości przypisanej do L-wartości.

Operator przypisania może wystąpić w instrukcji przypisania ale może także wystąpić wewnątrz wyrażeń, tak jak każdy inny operator, gdyż w przeciwieństwie do instrukcji posiada tę właściwość, że zwraca określony rezultat.

Operatory i wyrażenia arytmetyczne

operatory arytmetyczne	
`^`	potęgowanie
`*`	mnożenie
`/`	dzielenie (liczba zmiennoprzecinkowa)
`\`	dzielenie całkowitoliczbowe (liczba całkowita)
`Mod`	reszta z dzielenia całkowitoliczbowego
`+`	dodawanie
`-`	odejmowanie, zmiana znaku liczby

```
Option Explicit

Dim wynik As Single

wynik = 10 / 3 '3.333333 żzaleny od typu
```

```
Option Explicit

Dim wynik As Integer

wynik = 10 ^ 3    '1000
wynik = 10 * 3    '30
wynik = 10 \ 3    '3
wynik = 10 Mod 3  '1
wynik = 10 + 3    '13
wynik = 10 - 3    '7
wynik = 5         '5
wynik = -wynik    '-5
```

Kolejność (priorytety) wykonywania działań jest taka jak w matematyce, można ją regulować za pomocą nawiasów okrągłych. Operatory równoważne wykonywane są od strony lewej do prawej.

wybrane funkcje matematyczne	
`Sin()`	sinus
`Cos()`	cosinus
`Tan()`	tangens
`Atn()`	arcus tangent
`Abs()`	wartość bezwzględna
`Log()`	logarytm naturalny
`Exp()`	eksponent (e^x)

Operator konkatenacji

Operatorem konkatenacji (łączenia) są znaki dodawania + i ampersand &.

```
Dim imie, nazwisko, imieNazwisko As String

imie = "Ambroży"
nazwisko = "Kleks"

imieNazwisko = imie & " " & nazwisko
```

Instrukcja warunkowe, operatory relacji i wyrażenia logiczne

operatory relacji	
`=`	czy równe?
`<>`	czy różne?
`>`	czy większe?
`<`	czy mniejsze?
`>=`	czy większe lub równe?
`<=`	czy mniejsze lub równe?
`Like`	czy łańcuchy znaków takie same?
`Is`	porównanie zmiennych obiektowych

Operatory relacji mogą działać na wszystkich typach danych, jednak najczęściej odnoszą się do danych numerycznych i łańcuchów znaków.

Porównanie danych całkowitych nie powinno budzić zastrzeżeń, ponieważ operacje wykonywane są w sposób naturalny.

Należy zachować ostrożność przy porównywaniu wartości zmiennoprzecinkowych.

```
Dim pierwsza, druga As Double

pierwsza = 1.00000000000001
druga = 1.00000000000002

If pierwsza = druga then MsgBox("Równe!")

If Abs(pierwsza - druga) < 0.000001 then MsgBox("Równe!")
```

W przypadku łańcuchów wartość każdej litery w łańcuchu odnosi się do jej pozycji w alfabecie. Oznacza to, że litera **A** jest mniejsza od litery **B**, ta zaś od **C**, itd.

Wszystkie małe litery mają większą wartość niż ich duże odpowiedniki. Ponadto porównanie łańcuchów następuje kolejno, litera po literze zaczynając od lewej strony do prawej.

W przypadku operatora `Like` może zawierać znaki wieloznaczne: `?`, `*`, `#`.

Deklaracja:

```
Option Compare Binary
```

spowoduje, że operator `Like` będzie rozróżniał wielkość liter, natomiast

```
Option Compare Text
```

spowoduje, że operator `Like` nie będzie rozróżniał wielkości liter.

operatory logiczne	
`Not`	negacja (zaprzeczenie)
`And`	koniunkcja
`Or`	alternatywa
`Xor`	alternatywa wykluczająca

`Not`

argument	wynik
prawda	fałsz
fałsz	prawda

`And`

lewy argument	prawy argument	wynik
prawda	prawda	prawda
prawda	fałsz	fałsz
fałsz	prawda	fałsz
fałsz	fałsz	fałsz

`Or`

lewy argument	prawy argument	wynik
prawda	prawda	prawda
prawda	fałsz	prawda
fałsz	prawda	prawda
fałsz	fałsz	fałsz

`Xor`

lewy argument	prawy argument	wynik
prawda	prawda	fałsz
prawda	fałsz	fałsz
fałsz	prawda	fałsz
fałsz	fałsz	prawda

Wyrażenia logiczne mogą zawierać wyrażenia innego typu, np. arytmetyczne.

```
Dim wynik As Boolean

wynik = (5 < 3 * 2) And ("Lech" > "Adam")
wynik = (5 <> 3 * 2) Or ("Lech" = "Adam")
wynik = Not(2 + 2 = 5)
wynik = "Ala ma kota" Like "Ala ma*"
```

```
Dim liczba As Integer
Dim parzysta, zakres, wynik As Boolean

liczba = 25

parzysta = (liczba Mod 2) = 0
zakres = ((liczba >= 10) And (liczba <= 20))
wynik = parzysta And zakres
```

Instrukcja warukowa i wyboru

If - Then - End If

```
Dim a As Integer
Dim b As Integer

a = 1
b = 2

If (a + b) = 3 Then

  MsgBox("Podwojona suma równa się 6.")

End If
```

If - Then - Else - End If

```
Dim utarg As Single
Dim premia As Single

utarg = 29999.99

If utarg >= 30000 Then

  premia = 0.05 * utarg

Else

  premia = 0
  MsgBox("Premii nie będzie!")

End If
```

If - Then - ElseIf - Then - Else - End If

```
Dim a, b, c As Integer
Dim delta, x0, x1, x2 As Single

a = 1
b = 8
c = 2
delta = b^2 - 4 * a * c

If delta > 0 Then

  x1 = (-b - delta^0.5) / (2 * a)
  x2 = (-b + delta^0.5) / (2 * a)

ElseIf delta = 0 Then

  x0 = -b / (2 * a)

Else

  MsgBox("Brak pierwiastków rzeczywistych.")

End If
```

```
Dim zwierze As String

zwierze = InputBox("Zwierzę?")

If (zwierze = "Pies") Or (zwierze = "pies") Then

  MsgBox("Zwierzę to pies!")

ElseIf (zwierze = "Kot") Or (zwierze = "kot") Then

  MsgBox("Zwirzę to kot!")

Else

  MsgBox("Ni to pies, ni to kot...")

End If
```

```
Dim zwierze As String

zwierze = InputBox("Zwierzę?")

Select Case zwierze

  Case "Pies"
    MsgBox("Zwierzę to pies!")

  Case "Kot"
    MsgBox("Zwirzę to kot!")

  Case Else
    MsgBox("Ni to pies, ni to kot...")

End Select
```

Instrukcja z listami:

```
Dim zwierze As String

zwierze = InputBox("Zwierzę?")

Select Case zwierze

  Case "Pies", "pies"
    MsgBox("Zwierzę to pies!")

  Case "Kot", "kot"
    MsgBox("Zwirzę to kot!")

  Case Else
    MsgBox("Ni to pies, ni to kot...")

End Select
```

Instrukcja z przedziałami:

```
Dim strLiczba As String
Dim intLiczba As Integer

strLiczba = InputBox("Liczba?")
intLiczba = CInt(strLiczba)

Select Case intLiczba

  Case 1 To 9
    MsgBox("Liczba dodatnia jednocyfrowa!")

  Case 10 To 99
    MsgBox("Liczba dodatnia dwucyfrowa!")

  Case 0
    MsgBox("Zero to zero.")

  Case Else
    MsgBox("Jakaś inna liczba...")

End Select
```

Instrukcja z warunkami (Is):

```
Dim strLiczba As String
Dim intLiczba As Integer

strLiczba = InputBox("Liczba?")
intLiczba = CInt(strLiczba)

Select Case intLiczba

  Case Is <= 50
    MsgBox("Liczba jest mniejsza lub równa 50")

  Case Is 51
    MsgBox("Liczba jest równa 51")

  Case Is > 100
    MsgBox("Liczba jest większa od 100")

  Case Else
    MsgBox("Liczba jest w przedziale od 52 do 100")

End Select
```

Pętle

VBA oferuje dwa rodzaje pętli warunkowych:

- konstrukcja: `While` *warunek* `Wend`,
- konstrukcja `Do Loop`:
  - z warunkiem na początku petli:
    - `Do While` *warunek* `- Loop`,
    - `Do Until` *warunek* `- Loop`,
  - z warunkiem na końcu pętli:
    - `Do - Loop While` *warunek*,
    - `Do - Loop Until` *warunek*.

Pętle:

- `While` wykonują się **dopóki jest spełniony** warunek,
- `Until` wykonują się **aż zostanie spełniony** warunek.

While - Wend

```
Dim indeks As Integer
Dim licznik As Integer

indeks = 20
licznik = 0

While indeks > 10

  indeks = indeks - 1
  licznik = licznik + 1

Wend

MsgBox(licznik) 'licznik = 10
```

Do While - Loop

```
Dim indeks As Integer
Dim licznik As Integer

indeks = 20
licznik = 0

Do While indeks > 10

  indeks = indeks - 1
  licznik = licznik + 1

Loop

MsgBox(licznik) 'licznik = 10
```

Do Until - Loop

```
Dim indeks As Integer
Dim licznik As Integer

indeks = 20
licznik = 0

Do Until indeks > 10

  indeks = indeks - 1
  licznik = licznik + 1

Loop

MsgBox(licznik) 'licznik = 0
```

Do - Loop While

```
Dim indeks As Integer
Dim licznik As Integer

indeks = 20
licznik = 0

Do

  indeks = indeks - 1
  licznik = licznik + 1

Loop While indeks > 10

MsgBox(licznik) 'licznik = 10
```

Do - Loop Until

```
Dim indeks As Integer
Dim licznik As Integer

indeks = 20
licznik = 0

Do

  indeks = indeks - 1
  licznik = licznik + 1

Loop Until indeks > 10

MsgBox(licznik) 'licznik = 1
```

Przerwanie działanie pętli można osiągnąć na pomocą indtrukcji `Exit While` i `Exit Do`:

```
Dim indeks As Integer
Dim licznik As Integer

indeks = 20
licznik = 0

While indeks > 10

  indeks = indeks - 1
  licznik = licznik + 1

  If licznik = 5 Then Exit While

Wend

MsgBox(licznik) 'licznik = 5
```

Działanie pętli można **przerwać** za pomocą kombinacji klawiszy [CTRL BREAK].

For - To - Next

```
Dim i, n As Integer

n = 6

For i = 1 To n

  MsgBox(i)

Next 'i = 1 2 3 4 5 6
```

For - To - Step - Next

```
Dim i, n As Integer

n = 6

For i = 2 To n Step 2

  MsgBox(i)

Next 'i = 2 4 6
```

For - To - Step - Next

```
Dim i, n As Integer

n = 6

For i = n To 1 Step -2

  MsgBox(i)

Next 'i = 6 4 2
```

For - To - Step - Next

```
Option Base 1

Dim i, n As Single

n = 1.2

For i = 0 To n Step 0.4

  MsgBox(i)

Next 'i = 0 0.4 0.8 1.2
```

Przerwanie działanie pętli można osiągnąć na pomocą indtrukcji `Exit For`.

For Each - Next

```
Option Base 1

Dim imie As Variant, lista()
lista = Array("Ala", "Ola", "Ela")

For Each imie In lista

  MsgBox(imie)

Next
```

For - To - Next

```
Dim n As Integer
Dim imie As Variant
Dim lista()
lista = Array("Ala", "Ola", "Ela")

n = UBound(lista, 1)

For i = 1 To n

  imie = lista(i)
  MsgBox(imie)

Next
```

Komunikaty

Funkcja MsgBox

```
MsgBox(prompt[, buttons] [, title] [, helpfile, context])
```

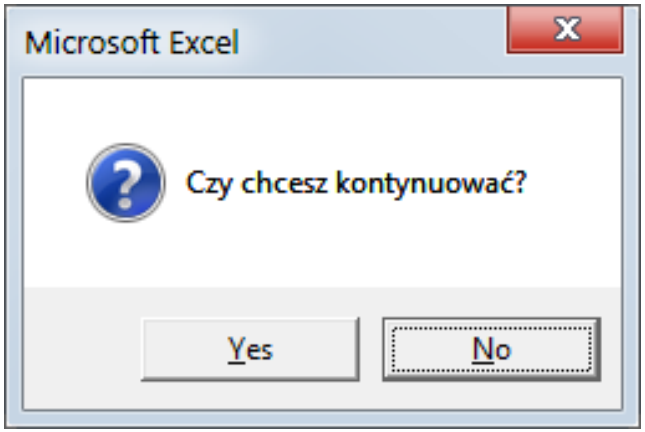
parametry MsgBox	
prompt	treść komunikatu (łańcuch znaków)
buttons	przyciski komunikatu (łańcuch znaków)
title	tytuł okna (łańcuch znaków)
helpfile	plik pomocy kontekstowej (łańcuch znaków)
context	numer tematu pomocy kontekstowej (wartość numeryczna)

wartości `buttons`		
`vbOKOnly`	0	tylko przycisk **OK**
`vbOKCancel`	1	przyciski **OK** i **Cancel**
`vbAbortRetryIgnore`	2	przyciski **Abort**, **Retry** i **Ignore**
`vbYesNoCancel`	3	przyciski **Yes**, **No** i **Cancel**
`vbYesNo`	4	przyciski **Yes** i **No**
`vbRetryCancel`	5	przyciski **Retry** i **Cancel**
`vbCritical`	16	ikona **alarm**
`vbQuestion`	32	ikona **pytanie**
`vbExclamation`	48	ikona **ostrzegawcza**
`vbInformation`	64	ikona **informacyjna**
`vbDefaultButton1`	0	przycisk pierwszy jest domyślny
`vbDefaultButton2`	256	przycisk drugi jest domyślny
`vbDefaultButton3`	512	przycisk trzeci jest domyślny
`vbDefaultButton4`	768	przycisk czwarty jest domyślny
`vbApplicationModal`	0	komunikat **modalny**

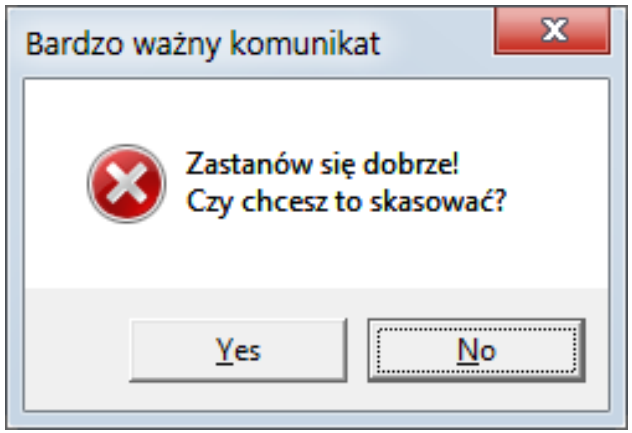
zwracane wartości		
`vbOK`	1	przycisk **OK**
`vbCancel`	2	przycisk **Cancel**
`vbAbort`	3	przycisk **Abort**
`vbRetry`	4	przycisk **Retry**
`vbIgnore`	5	przycisk **Ignore**
`vbYes`	6	przycisk **Yes**
`vbNo`	7	przycisk **No**

```vba
Dim wynik As Integer

wynik = MsgBox("Czy chcesz kontynuować?", vbYesNo + vbQuestion + vbDefaultButton2)
```

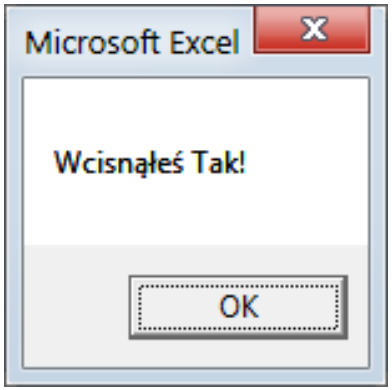
```vba
Dim wynik As Integer

wynik = MsgBox("Czy chcesz kontynuować?", 292)
```

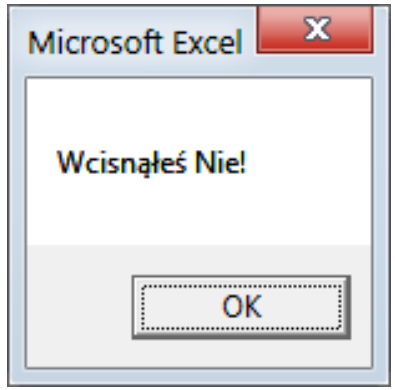


```
Dim tytul, komunikat As String
Dim styl, wynik As Integer

tytul = "Bardzo ważny komunikat"
styl = vbYesNo + vbCritical + vbDefaultButton2
komunikat = "Zastanów się dobrze!" & Chr(13) & "Czy chcesz to skasować?"

wynik = MsgBox(komunikat, styl, tytul)
```



```
Dim tytul, komunikat As String
Dim styl, wynik As Integer

tytul = "Bardzo ważny komunikat"
styl = vbYesNo + vbCritical + vbDefaultButton2
komunikat = "Zastanów się dobrze!" & Chr(13) & "Czy chcesz to skasować?"

wynik = MsgBox(komunikat, styl, tytul)

If wynik = vbYes Then
  MsgBox ("Wcisnąłeś Tak!")
Else
  MsgBox ("Wcisnąłeś Nie!")
End If
```



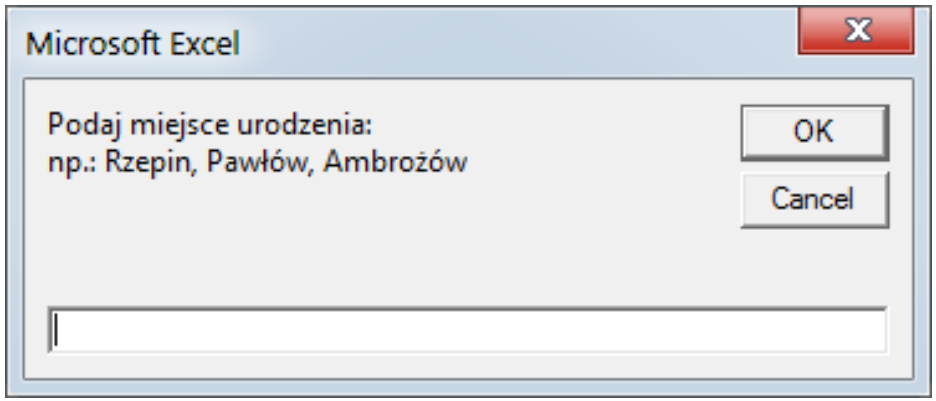


Funkcja InputBox

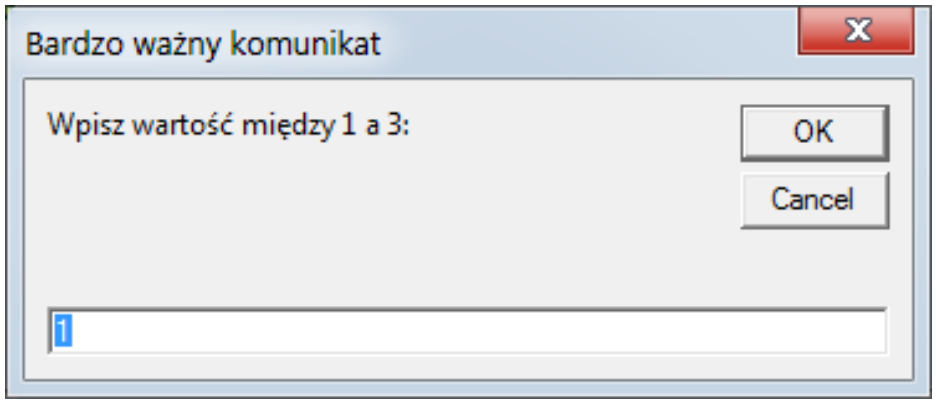
```
InputBox(prompt[, title] [, default] [, xpos] [, ypos] [, helpfile, context])
```

parametry MsgBox	
prompt	treść komunikatu (łańcuch znaków)
title	tytuł okna (łańcuch znaków)
default	domyślna odpowiedź (łańcuch znaków)
xpos	odległość od lewej krawędzi ekranu (wartość numeryczna)
ypos	odległość od górnej krawędzi ekranu (wartość numeryczna)
helpfile	plik pomocy kontekstowej (łańcuch znaków)
context	numer tematu pomocy kontekstowej (wartość numeryczna)

```
Dim miasto As String

miasto = InputBox("Podaj miejsce urodzenia:" & Chr(13) _
  & "np.: Rzepin, Pawłów, Ambrożów")
```

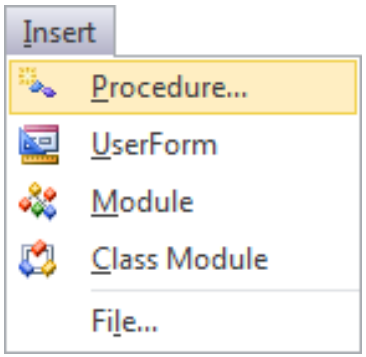


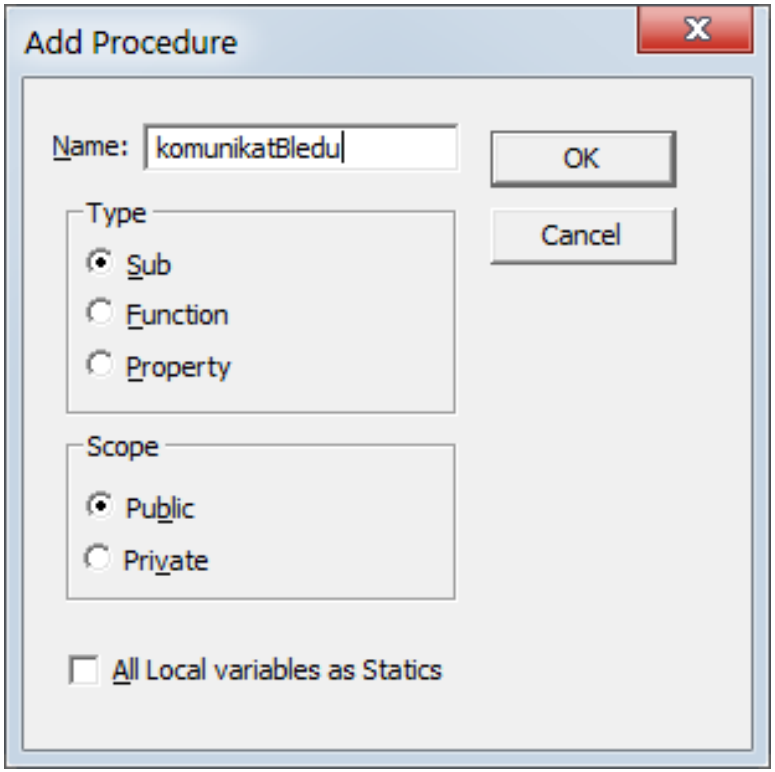
```
Dim tytul, komunikat, wynik, domyslna As String

komunikat = "Wpisz wartość między 1 a 3:"
tytul = "Bardzo ważny komunikat"
domyslna = "1"

wynik = InputBox(komunikat, tytul, domyslna, 200, 200)
```



Procedury

Procedurę definiuje się używając słów kluczowych `Sub` i `End Sub`. Można je dodawać w edytorze VB (menu **Insert**). Makropolecenia rejestrowane w Excelu są właśnie procedurami.





Przykład:

```
Sub komunikatBledu()

  MsgBox("Wystąpił błąd!")

End Sub
```

Przekazywanie parametru, typ (`Variant`):

```
Sub komunikatBledu(numerBledu)

  MsgBox("Wystąpił błąd nr " & CStr(numerBledu) & "!")

End Sub
```

Domyślnie, parametry są przekazywane z modyfikatorem ByRef:

```
Dim numer As Byte

numer = 3
komunikatBledu numerBledu:=numer
MsgBox(numer) '5

numer = 3
komunikatBledu(numer)
MsgBox(numer) '3

Sub komunikatBledu(numerBledu As Byte)
'   komunikatBledu(ByRef numerBledu As Byte)

  MsgBox("Wystąpił błąd nr " & CStr(numerBledu) & "!")
  numerBledu = 5

End Sub
```

Modyfikator ByVal:

```
Dim numer As Byte

numer = 3
komunikatBledu numerBledu:=numer
MsgBox(numer) '3

numer = 3
komunikatBledu(numer)
MsgBox(numer) '3

Sub komunikatBledu(ByVal numerBledu As Byte)

  MsgBox("Wystąpił błąd nr " & CStr(numerBledu) & "!")
  numerBledu = 5

End Sub
```

Funkcje

Funkcję definiuje się używając słów kluczowych `Function` i `End Function`. Można je dodawać w edytorze VB (menu **Insert**).

```
Dim wartX, wartY, wynik As Integer

wartX = 2
wartY = 3

wynik = mnozenie(wartX, wartY) '6

Function mnozenie(ByVal wartA As Integer, ByVal wartB As Integer) As Integer

  mnozenie = wartA * wartB

End Function
```

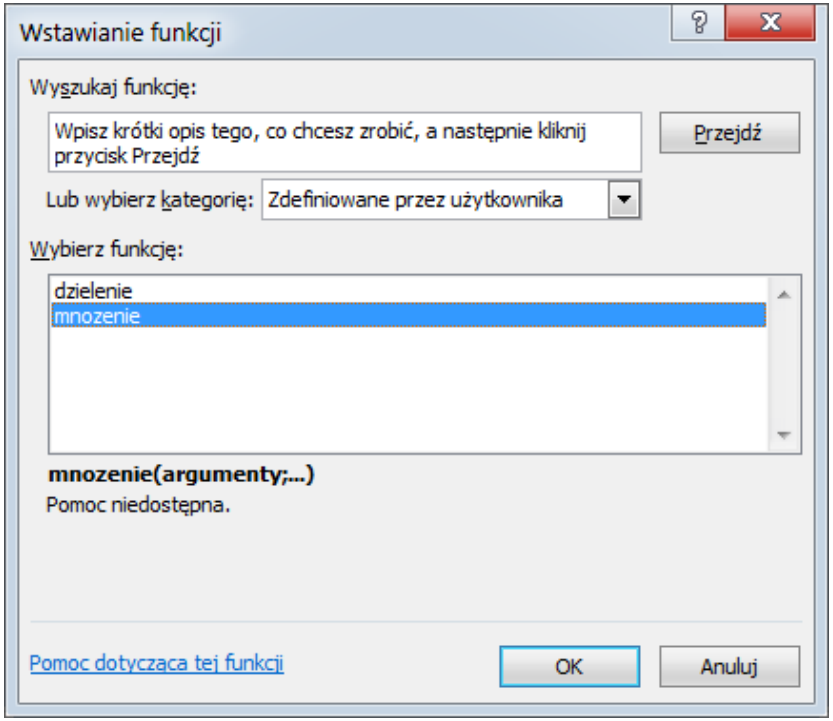
Parametr opcjonalny:

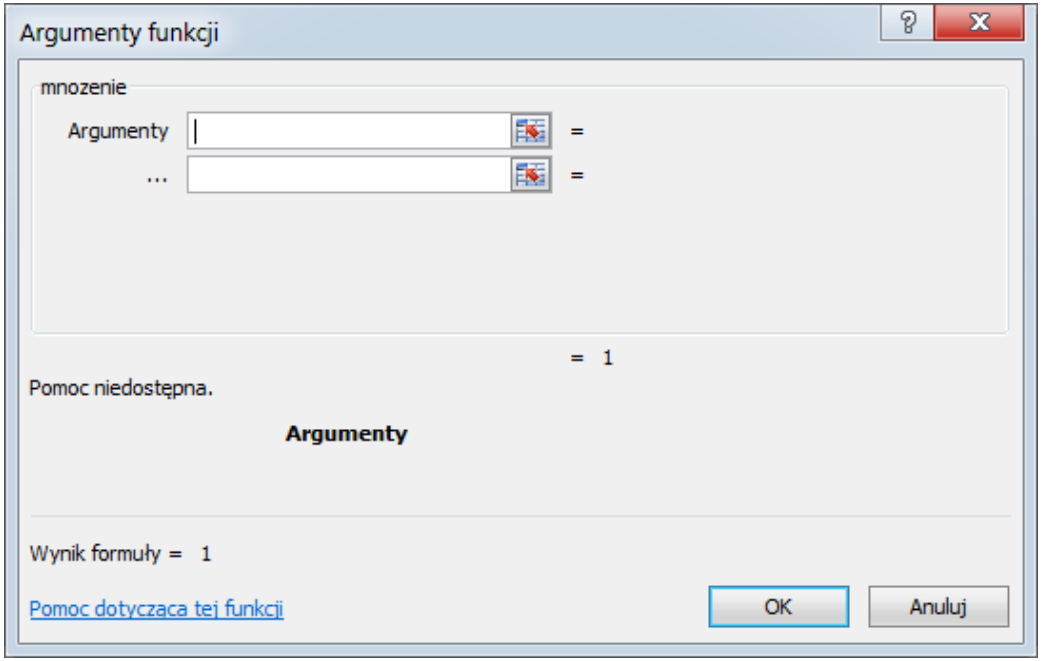
```
Dim wartX, wartY, wynik As Integer

wartX = 2
wartY = 3

wynik = mnozenie(wartX, wartY) '6
wynik = mnozenie(wartX, wartY, 5) '30

Function mnozenie(ByVal wartA As Integer, _
  ByVal wartB As Integer, Optional wartC) As Integer

  If IsMissing(wartC) Then

    mnozenie = wartA * wartB

  Else

    mnozenie = wartA * wartB * wartC

  End If

End Function
```

Lista parametrów:

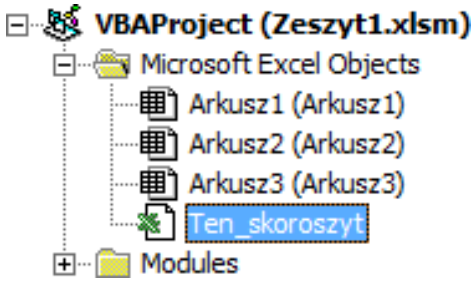
```
Dim wynik As Integer

wynik = mnozenie(1, 2) '2
wynik = mnozenie(1, 2, 3) '6
wynik = mnozenie(1, 2, 3, 4) '24

Function mnozenie(ParamArray argumenty()) As Integer

  Dim wartosc As Variant

  mnozenie = 1

  For Each wartosc In argumenty

    mnozenie = mnozenie * wartosc

  Next wartosc

End Function
```

Tak zdefiniowanych funkcji można używać w arkuszu.





Procedura prywatna skoroszytu o nazwie `Workbook_Open()` będzie automatycznie uruchamiana podczas otwierania arkusza.

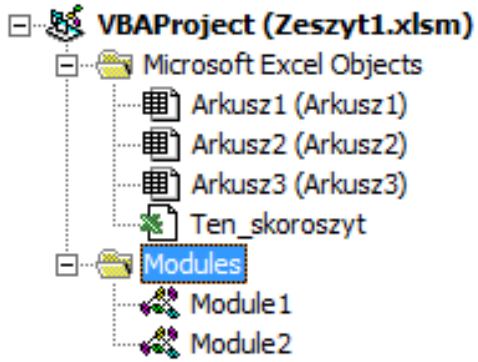



```
Private Sub Workbook_Open()

  MsgBox ("Otwarto arkusz.")

End Sub
```

Modyfikatory

Podczas omawiania zmiennych wspomniany został modyfikator `Public`. Przy definiowania zmiennych i podprogramów VBA pozwala na stosownie modyfikatorów:

- `Public`,
- `Private`,
- `Friend`,
- `Static`.



modyfikatory	
`Public`	podprogram jest dostępny dla wszystkich innych podprogramów z projektu
`Private`	ogranicza dostęp tylko dla podprogramów z modułu
`Friend`	podprogram będzie dostępny w projekcie, ale nie będzie można go używać np. z innego arkusza, (tylko obiekty)
`Static`	wartości zmiennych podprogramu będą przechowywane do następnego wywołania

Obsługa błędów

Instrukcja `On Error` pozwala na obsługę błędów zaistniałych w trakcie działania programu.

Umieszcza się ją przed blokiem instrukcji, który ma być kontrolowany:

```
Dim wartA, wartB, wynik As Single

wartA = 1
wartB = 0

On Error GoTo bladDzielenia

wynik = wartA / wartB
MsgBox ("wynik = " & wynik)

bladDzielenia:
  MsgBox ("Błąd.")
```

Instrukcja `On Error` zwykle występuje z `Exit Sub` lub `Exit Function`, zapobiega to wykonania instrukcji obsługi błędu, jeśli błąd nie wystąpił:

```
Function dzielenie(ByVal wartA As Double, ByVal wartB As Double) As Double

  On Error GoTo bladDzielenia

  dzielenie = wartA / wartB
  Exit Function

  bladDzielenia:
  MsgBox ("Błąd.")

End Function
```

Instrukcja Resume pozwala na powrót do instrukcji w której wystąpił błąd:

```
Function dzielenie(ByVal wartA As Double, ByVal wartB As Double) As Double

  On Error GoTo bladDzielenia

  dzielenie = wartA / wartB
  Exit Function

  bladDzielenia:
    wartB = 1
    Resume

End Function
```

Instrukcja występuje jeszcze w wersjach:

- `On Error Resume Next` – powoduje skok do następnej instrukcji, po instrukcji w której wystąpił błąd,
- `On Error Goto 0` – powoduje wyłączenie obsługi błędów,
- `On Error Goto -1` – powoduje skok do etykiety określonej w ostatnim wywołaniu `On Error Goto`.

Uwagi

Ze względu na ograniczony czas wykładu, nie poruszone zostały następujące, choć dość ważne zagadnienia:

- obiektowy charakter VBA (*Class Modules*),
- hierarchia obiektów specyficznych dla programu Excel ,
- tworzenie i obsługa okien (form),
- obsługa plików,
- zagadnień związanych z realizacją i uruchamianiem projektów,
  a w szczególności: krokowego wykonywania programów, pracy z punktami przerwania, śledzenia wartości zmiennych.