MINUT 2025 (7), s. 1–55
ISSN 2719-3063

Damian SŁOTA

Katedra Metod Matematycznych w Technice i Informatyce, Wydzial Matematyki Stosowanej, Politechnika Śląska, ul. Kaszubska 23, 44-100 Gliwice

Elementy języka Wolfram

Streszczenie. W prezentowanych materiałach dydaktycznych przedstawiono podstawy języka *Wolfram* pakietu *Mathematica*. W szczególności opisano wykorzystywane w pakiecie systemy liczbowe oraz różne instrukcje do ich obsługi. Przedstawiono instrukcje warunkowe oraz instrukcje pętli. Opisano sposób definiowania funkcji i procedur. Omówiono mechanizmy opcji oraz atrybutów, a także pracę z argumentami. Przedstawiono również podstawowe instrukcje wejścia-wyjścia.

Słowa kluczowe: Mathematica, CAS, język Wolfram.

Wstęp

Wolfram jest językiem programowania pakietu *Mathematica* firmy Wolfram Research, Inc. Pierwsza wersja pakietu *Mathematica* ukazała się 23 czerwca 1988 r. Obecnie dystrybuowana jest wersja 14.2. Wyłącznym dystrybutorem programów firmy Wolfram Research, Inc. na terenie Polski jest firma Gambit (mathematica.pl, gambit.net.pl).

Mathematica jest znakomitym środowiskiem zarówno do obliczeń symbolicznych, jak i do opracowywania danych eksperymentalnych, obliczeń przybliżonych czy tworzenia grafiki matematycznej. Łatwość obsługi, intuicyjność oraz bardzo duże możliwości rekomendują program *Mathematica* jako doskonałe środowisko pracy dla studentów i uczniów, jak i pracowników naukowych, stosujących w pracach badawczych zaawansowany aparat matematyczny [5]. W wielu zastosowaniach istotna będzie możliwość bezpośredniego korzystania z bazy wiedzy *Wolfram Knowledgebase* utworzonej przez firmę Wolfram Research. *Mathematica* umożliwia obliczenia w chmurze, za pośrednictwem przeglądarki internetowej, jak również lokalnie na komputerach.

Więcej informacji na temat pakietu *Mathematica* oraz innych produktów firmy Wolfram Research, Inc. można znaleźć na stronie wolfram.com. Można tam znaleźć m.in. przygotowane przez użytkowników projekty (Wolfram Demonstrations Project), a także nowe instrukcje (Wolfram Function Repository).

Niniejsze materiały dydaktyczne są kontynuacją materiałów, w których omówiono podstawy pakietu *Mathematica* [4]. Pierwsza wersja prezentowanych materiałów powstała wiele lat temu na potrzeby prowadzonych szkoleń. Z biegiem lat były one modyfikowane i poprawiane wraz z rozwojem języka *Wolfram*.

Autor korespondencyjny: D.Słota (damian.slota@polsl.pl).
Data wpłynięcia: 10.01.2025 r.

Materiały były także wykorzystane na wykładzie z przedmiotu platformy obliczeniowe prowadzonego dla studentów Wydziału Matematyki Stosowanej Politechniki Śląskiej. Ostatnie szkolenie odbyło się we wrześniu 2024 roku w ramach projektu „F2S Politechnika Śląska uczelnią wspierającą kadrę w drodze do doskonałości". Materiały zostały opracowane w postaci notatnika (plik *.nb) pakietu *Mathematica*. Na dalszych stronach niniejszego pliku zawarty jest wydruk do pliku pdf przygotowanego notatnika. Dlatego też styl następnych stron trochę się różni od standardów czasopisma MINUT. Na stronie czasopisma oprócz pliku pdf dostępny jest także plik notatnika (nb). Notatnik może być rozpowszechniany zgodnie z licencją CC BY-NC-SA 4.0. Obliczenia wykonano w wersji 14.2 pakietu. Notatnik zawiera podstawowe instrukcje, które będą działać także w wielu wcześniejszych wersjach pakietu.

Literatura

1. H. Gliński, R. Grzymkowski, A. Kapusta, D. Słota, *Mathematica 8*, WPKJS, Gliwice 2012.
2. R. Grzymkowski, A. Kapusta, T. Kuboszek, D. Słota, *Mathematica 6*, WPKJS, Gliwice 2008.
3. J.A. Miszczak, *Short Lessons on Wolfram Language*, Quoqa Pubs, Knurów 2024.
4. D. Słota, *Mathematica – podstawowe instrukcje*, MINUT **6** (2024), 92–161.
5. Strona: *mathematica.pl*, dostęp 10.01.2025.
6. S. Wolfram, *The Mathematica Book*, 5th ed., Wolfram Media/Cambridge University Press, Champaign/Cambridge 2003.
7. S. Wolfram, *An Elementary Introduction to the Wolfram Language*, Wolfram Media, Champaign 2017.

1. Liczby, zmienne i operatory

1.1. Zbiory liczbowe

In[1]:=
```
Clear["Global`*"]
```

W programie mamy do dyspozycji stałe reprezentujące odpowiednie zbiory liczbowe oraz zbiór wartości logicznych, którego elementami są dwie stałe: True - reprezentująca wartość logiczną „prawda” oraz False - reprezentująca wartość logiczną „fałsz”.

```
 Integers          zbiór liczb całkowitych;
Rationals          zbiór liczb wymiernych;
  Reals           zbiór liczb rzeczywistych;
Complexes          zbiór liczb zespolonych;
Algebraics        zbiór liczb algebraicznych;
  Primes            zbiór liczb pierwszych;
 Booleans  zbiór wartości logicznych (True, False).
```

Jeśli chcemy sprawdzić, czy liczba należy do odpowiedniego zbioru, musimy wykorzystać symbol przynależności ∈ (dostępny na palecie *Basic Math Input*) lub instrukcję Element.

```
x∈Y lub Element[x, Y]   sprawdza, czy podana liczba x należy do zbioru Y .
```

Sprawdźmy, czy liczba 123456789 jest liczbą pierwszą:

In[2]:=
```
123 456 789 ∈ Primes
```

Out[2]=
```
False
```

Odpowiedź False oznacza, że badana liczba nie jest liczbą pierwszą.

In[3]:=
```
Element[252 097 800 623, Primes]
```

Out[3]=
```
True
```

Zobaczmy także do jakich zbiorów należy liczba 10.

In[4]:=
```
{10 ∈ Integers, 10 ∈ Rationals, 10 ∈ Reals, 10 ∈ Primes}
```

Out[4]=
```
{True, True, True, False}
```

W programie *Mathematica* wartość logiczne są oznaczane przez True i False, a nie przez 0 i 1.

In[5]:=
```
{True ∈ Booleans, False ∈ Booleans, 1 ∈ Booleans, 0 ∈ Booleans}
```

Out[5]=
```
{True, True, False, False}
```

1.2. Typy liczb

In[6]:=
```
Clear["Global`*"]
```

W obliczeniach w programie wykorzystywane są cztery typy liczb.

```
 Integer   liczby całkowite;
Rational   liczby wymierne;
  Real    liczby rzeczywiste;
 Complex   liczby zespolone.
```

Liczby postaci 2, 6, 23 itp. są interpretowane jako liczby całkowite, natomiast liczby z kropką dziesiętną (np. 2.5, 2., 2.0) są interpretowane jako liczby rzeczywiste. Liczby wymierne są zapisywane w postaci ułamków $\frac{p}{q}$, gdzie p i q są liczbami całkowitymi, $q \neq 0$ i p nie jest podzielne przez q (w przypadku, gdy jest podzielne, to liczba jest typu całkowitego). Jako liczby zespolone są interpretowane liczby postaci $2+3i$, $2+0i$ itp. Jeśli zachodzi taka potrzeba, w trakcie obliczeń dokonywana jest automatyczna konwersja typów liczb (do typu największego, względem relacji zawierania, zbioru liczb wykorzystywanego w obliczeniach). Typ liczby możemy sprawdzić instrukcją Head.

```
Head[liczba] podaje typ liczby.
```

Sprawdźmy jakiego typu są liczby 2 , 2., $\frac{4}{2}$ i $\frac{5}{2}$.

In[7]:=
```
{Head[2], Head[2.], Head[4/2], Head[5/2]}
```

Out[7]=
```
{Integer, Real, Integer, Rational}
```

1.3. Instrukcje sprawdzające „rodzaj" liczby

In[8]:=
```
Clear["Global`*"]
```

Dostępne są także instrukcje pozwalające sprawdzić, czy podana liczba jest liczbą parzystą, nieparzystą, dodatnią, pierwszą itp.

```
   IntegerQ[x]       sprawdza, czy podana liczba jest liczbą całkowitą;
     EvenQ[x]         sprawdza, czy podana liczba jest liczbą parzystą;
      OddQ[x]        sprawdza, czy podana liczba jest liczbą nieparzystą;
    PrimeQ[x]         sprawdza, czy podana liczba jest liczbą pierwszą;
   Positive[x]        sprawdza, czy podana liczba jest liczbą dodatnią;
   Negative[x]         sprawdza, czy podana liczba jest liczbą ujemną;
 NonPositive[x]     sprawdza, czy podana liczba jest liczbą niedodatnią;
 NonNegative[x]      sprawdza, czy podana liczba jest liczbą nieujemną;
NumberQ[zmienna]  sprawdza, czy podana zmienna jest liczbą dowolnego typu;
NumericQ[zmienna] sprawdza, czy podana zmienna posiada wartość numeryczną.
```

Sprawdźmy, czy liczba 2025 jest liczbą pierwszą.

In[9]:=
```
PrimeQ[2025]
```

Out[9]=
```
False
```

Liczba zero nie jest zaliczana do zbiorów liczb dodatnich lub ujemnych, natomiast należy do zbiorów liczb nieujemnych i niedodatnich.

In[10]:=
```
{Positive[0], Negative[0], NonNegative[0], NonPositive[0]}
```

Out[10]=
```
{False, False, True, True}
```

Różnica pomiędzy ostatnimi dwoma instrukcjami ujawnia się w przypadku zdefiniowanych w programie stałych matematycznych i fizycznych. W programie *Mathematica* oznaczenie stałej (np. π, e) jest traktowane jako symbol, który posiada wartość numeryczną, a nie jako liczba określonego typu.

In[11]:=
```
{NumberQ[π], NumericQ[π]}
```

Out[11]=
```
{False, True}
```

1.4. Wartości przybliżone

In[12]:=
```
Clear["Global`*"]
```

Możemy także wykorzystywać instrukcje, które umożliwiają obliczenie przybliżonej wartości wyrażenia lub znalezienie liczby wymiernej przybliżającej podaną liczbę.

```
N[wyrażenie] lub wyrażenie // N  podaje przybliżoną wartość wyrażenia;
       N[wyrażenie, n]           podaje przybliżoną wartość wyrażenia
                                  z dokładnością do n cyfr znaczących;
     Rationalize[liczba]         podaje liczbę wymierną przybliżającą
                                 podaną liczbę;
Rationalize[liczba, dokładność]  podaje liczbę wymierną przybliżającą
                                  podaną liczbę z zadaną dokładnością.
```

Wypiszmy wartość numeryczną stałej π z dokładnością do dwudziestu cyfr znaczących, czyli istotnych cyfr przed przecinkiem i po przecinku.

```
In[13]:= N[π, 20]

Out[13]= 3.1415926535897932385
```

W wyniku otrzymaliśmy liczbę, w której przed przecinkiem jest jedna cyfra, natomiast po przecinku dziewiętnaście cyfr, czyli w sumie liczba cyfr przed przecinkiem i po przecinku jest równa dwadzieścia.

Zobaczmy inne przykłady.

```
In[14]:= N[10 * π, 50]

Out[14]= 31.415926535897932384626433832795028841971693993751

In[15]:= N[π / 10, 200]

Out[15]= 0.314159265358979323846264338327950288419716939937510582097494459230781640628620899862803482
          534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555
          96446229489549303820
```

Liczba wyświetlanych cyfr może być bardzo duża (ograniczeniem w tym zakresie jest jedynie dostępna pamięć oraz sprzęt, na jakim jest zainstalowany program).

```
In[16]:= N[ⅇ, 500]

Out[16]= 2.718281828459045235360287471352662497757247093699959574966967627724076630353547594571382178
          525166427427466391932003059921817413596629043572900334295260595630738132328627943490763233
          829880753195251019011573834187930702154089149934884167509244761460668082264800168477411853
          742345442437107539077744992069551702761838606261331384583000752044933826560297606737113200
          709328709127443747047230696977209310141692836819025515108657463772111252389784425056953696
          7707854499699679468644549059879316368892300987931
```

Liczby wymierne zapisane w postaci ułamka dziesiętnego, możemy przedstawić w postaci ułamka zwykłego wykorzystując instrukcję Rationalize.

```
In[17]:= Rationalize[1.315]
```

Out[17]= $\frac{263}{200}$

Jeśli chcemy przybliżyć liczbę niewymierną, to musimy podać dokładność tego przybliżenia.

```
In[18]:= Rationalize[π]

Out[18]= π
```

In[19]:= r = Rationalize$\left[\pi, 10^{-3}\right]$

Out[19]= $\frac{201}{64}$

```
In[20]:= {r, π, Abs[r - π]} // N

Out[20]= {3.14063, 3.14159, 0.000967654}
```

1.5. Konwersja między różnymi systemami liczbowymi

```
In[21]:= Clear["Global`*"]
```

Następne dwie instrukcje umożliwiają dokonywanie konwersji liczb zapisanych w systemie dziesiętnym na inny system liczbowy lub odwrotnie.

BaseForm[liczba, podstawa]	podaje zapis danej liczby w systemie liczbowym o podanej podstawie;
podstawa ^^ liczba	podaje zapis w systemie dziesiętnym liczby zapisanej w systemie liczbowym o podanej podstawie.

Zapiszmy liczbę 2025 w systemie dwójkowym.

```
In[22]:= BaseForm[2025, 2]
```

$$\text{Out[22]//BaseForm=}\quad 11111101001_2$$

Indeks na końcu liczby (w tym przypadku liczba 2) oznacza system liczbowy, w jakim jest zapisana dana liczba.

Dokonajmy konwersji z sytemu binarnego (dwójkowego) na system dziesiętny.

```
In[23]:= 2^^10111101
Out[23]= 189
```

Możemy także wykonywać konwersje na inne systemy liczbowe.

```
In[24]:= BaseForm[2025, 16]
```

$$\text{Out[24]//BaseForm=}\quad 7e9_{16}$$

1.6. Operacje arytmetyczne

```
In[25]:= Clear["Global`*"]
```

Mathematica pozwala wykonywać obliczenia tak jak na kalkulatorze, z tym, że zamiast znaku mnożenia (*) można używać spacji. Użycie spacji lub znaku mnożenia jest konieczne, zapis xy oznacza coś innego niż iloczyn, a mianowicie nową zmienną o nazwie xy. W programie dostępne są (często w kilku formach) podstawowe operacje matematyczne.

```
   x + y lub Plus[x, y]          dodawanie;
         x - y                  odejmowanie;
   -x lub Times[-1, x]      element przeciwny;
x * y lub x y lub Times[x, y]     mnożenie;
   x / y lub Divide[x, y]        dzielenie;
x^y lub x^y lub Power[x, y]     potęgowanie.
```

Jeśli dzielimy liczby całkowite, to wynik otrzymujemy w postaci ułamka zwykłego, gdyż *Mathematica*, jako program ukierunkowany na obliczenia symboliczne, przedstawia wynik w postaci dokładnej (chyba że zażądamy obliczeń przybliżonych).

```
In[26]:= 342 / 21
```

$$\text{Out[26]=}\quad \frac{114}{7}$$

Jeśli chcemy uzyskać wynik w postaci ułamka dziesiętnego, to możemy zastosować między innymi jedną z metod podanych poniżej.

```
In[27]:= 342. / 21
Out[27]= 16.2857
In[28]:= 342 / 21 // N
Out[28]= 16.2857
In[29]:= N[342 / 21]
Out[29]= 16.2857
```

W przypadku opisanych powyżej operacji, wartości zmiennych x i y nie ulegają zmianie, dlatego operacje te mogą być wykonywane także na stałych. Poniżej przedstawiamy z kolei operacje, w wyniku których wartość pierwszej ze zmiennych ulega zmianie. W tym wypadku pierwszy argument (stojący z lewej strony znaku operacji) musi być zmienną, drugi może być stałą.

```
   x += y lub AddTo[x, y]            dodaje do wartości zmiennej x wartość y
                                       (jest równoważny zapisowi x = x + y);
x -= y lub SubtractFrom[x, y] odejmuje od wartości zmiennej x wartość y (x = x - y);
   x *= y lub TimesBy[x, y]     mnoży wartość zmiennej x przez wartość y (x = x * y);
  x /= y lub DivideBy[x, y]      dzieli wartość zmiennej x przez wartość y (x = x/y).
```

Możemy dodawać wartości stałe,

```
In[30]:= 2 + 3
Out[30]= 5
```

ale zmieniać wartość możemy tylko w przypadku zmiennej.

```
In[31]:= x = 0; Table[x += 3, {i, 1, 4}]
Out[31]= {3, 6, 9, 12}
```

Ostatnią grupę operacji stanowią operacje jednoargumentowe zwiększające lub zmniejszające wartość danej zmiennej o jeden, czyli instrukcje inkrementacji i dekrementacji.

```
  x++ lub Increment[x]
  x-- lub Decrement[x]
++x lub PreIncrement[x]
--x lub PreDecrement[x]
```

Widzimy powyżej po dwie formy operacji inkrementacji i dekrementacji (przedrostkowa ++x i końcówkowa x++). Różnica między nimi polega na tym, że jeśli operator inkrementacji (dekrementacji) stoi przed zmienną (++x, --x), to w czasie obliczania instrukcji zawierającej takie wyrażenie, najpierw wartość zmiennej jest zwiększana (zmniejszana) o jeden, a dopiero potem następuje wykonanie pozostałej części instrukcji. Natomiast w drugim przypadku, gdy operator stoi za zmienną (x++ lub x--), to najpierw jest wykonywana instrukcja (ze starą wartością zmiennej), a dopiero potem jest zmieniana wartość zmiennej. Zobaczmy, jak wygląda to w praktyce.

```
In[32]:= x = 2; y = x++; {x, y}
Out[32]= {3, 2}
In[33]:= x = 2; y = ++x; {x, y}
Out[33]= {3, 3}
```

Ostatnie dwie grupy operacji są bardzo często używane podczas pisania programów. Upraszczają one zapis, a przede wszystkim skracają czas ich wykonania (w stosunku do programów z zapisami typu: x=x+1).

1.7. Operatory logiczne i relacje

```
In[34]:= Clear["Global`*"]
```

Ułatwieniem przy sprawdzaniu różnych warunków i założeń jest możliwość zastosowania relacji. Sposoby zapisu relacji przedstawione są poniżej.

```
   x == y lub Equal[x, y] lub z palety ==          równe;
  x ≠ y lub Unequal[x, y] lub z palety ≠    nierówne (różne);
        x > y lub Greater[x, y]                 większe;
x ≥ y lub GreaterEqual[x, y] lub z palety ≥ większe lub równe;
          x < y lub Less[x, y]                  mniejsze;
 x ≤ y lub LessEqual[x, y] lub z palety ≤  mniejsze lub równe;
        x === y lub SameQ[x, y]               identyczne;
       x =!= y lub UnsameQ[x, y]            nieidentyczne.
```

Wszystkie relacje możemy łączyć w ciągi postaci:

$$x_0\ R_1\ x_1\ R_2\ \ldots\ R_n\ x_n,$$

gdzie R_i jest dowolną z relacji opisanych powyżej. Sprawdźmy, czy kolejne dwie wartości są różne:

```
In[35]:= 0^2 ≠ 0 ≠ 0!
Out[35]= False
```

oraz czy prawdziwy jest poniższy ciąg relacji.

```
In[36]:= 2^5 > 19 < 20 ≥ 20 > 0 ≤ 1
Out[36]= True
```

W przypadku, gdy zmienne nie mają nadanej wartości, *Mathematica* nie jest w stanie rozstrzygnąć czy formuła jest prawdziwa, czy fałszywa.

In[37]:=
```
x1 ≤ y1
```

Out[37]=
```
x1 ≤ y1
```

Przyjrzyjmy się teraz operacjom logicznym, które można stosować w programie.

```
   ! p lub Not[p] lub z palety ¬      negacja;
p && q lub And[p, q] lub z palety ∧  koniunkcja;
 p || q lub Or[p, q] lub z palety ∨ alternatywa;
        Implies[p, q]               implikacja
```

Dla operacji logicznych możemy tworzyć ciągi, w taki sam sposób jak dla relacji. Sprawdźmy wartość koniunkcji.

In[38]:=
```
2 ≤ 3 ∨ 5 < 0
```

Out[38]=
```
True
```

Jeśli nie można określić, czy wyrażenie jest prawdziwe, to zostanie ono wyprowadzone bez zmian.

In[39]:=
```
(p ∨ q) ∧ (r ∨ p)
```

Out[39]=
```
(p || q) && (r || p)
```

1.8. Definiowanie zmiennych

In[40]:=
```
Clear["Global`*"]
```

Podczas długotrwałych obliczeń przydatne jest oznaczanie pośrednich wyników poprzez nadanie im nazwy.

```
   x = wartość          przypisanie wartości zmiennej x;
 x = y = wartość   przypisanie tej samej wartości zmiennym x i y;
x =. lub Clear[x]  anulowanie wartości przypisanej zmiennej x.
```

Poprzez przypisanie wartości zmiennej możemy parametryzować obliczenia. Przypisanie jest ważne aż do jego anulowania. Zdefiniujmy dwie zmienne.

In[41]:=
```
x = 5; y = 6;
```

Zobaczmy, czy definicja została zapamiętana.

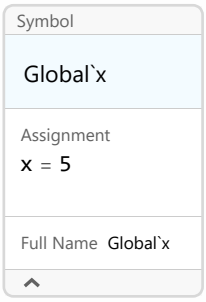
In[42]:=
```
{x, y}
```

Out[42]=
```
{5, 6}
```

Informację o zdefiniowanej zmiennej możemy uzyskać także korzystając z instrukcji „?”.

In[43]:=
```
?x
```

Out[43]=



A teraz w oparciu o nie określmy następną zmienną.

In[44]:=
```
z = x + y - 3.1
```

Out[44]=
```
7.9
```

Anulujmy wartości nadane zmiennym x i y.

```
In[45]:= Clear[x, y]
```

Ponieważ instrukcja Clear nie generuje żadnego wyniku, więc nie pojawia się komórka wyjściowa (Out[]). Sprawdźmy, jaką teraz dostaniemy odpowiedź na pytanie o wartość zmiennej x.

```
In[46]:= x

Out[46]= x
```

Instrukcja „?” wypisze teraz tylko informację, że symbol był wykorzystywany w środowisku Global.

```
In[47]:= ?x
```

```
Out[47]=
Symbol
Global`x
Full Name Global`x
```

Dla symbolu, który nie był wykorzystywany dostaniemy następującą informację:

```
In[48]:= ?x16

Out[48]= Missing[UnknownSymbol, x16]
```

Zapoznajmy się teraz z kilkoma uwagami dotyczącymi interpretacji wyrażeń przez program, pozwoli nam to uniknąć w przyszłości poprawiania błędnie skonstruowanych poleceń. Oto niektóre z wyrażeń i ich objaśnienia:

```
  x y        oznacza x razy y;
  xy   oznacza zmienną o nazwie xy;
  2 x        oznacza 2 razy x;
x^2 y  oznacza (x^2) y, a nie x^(2 y).
```

Zmiennej symbolicznej (bez nadanej wartości) występującej w dowolnym wyrażeniu możemy nadać konkretną wartość. Możemy to wykonać w następujący sposób:

```
     wyrażenie /. x → wartość      podstawienie w wyrażeniu w miejsce zmiennej x podanej wartości;
    wyrażenie /. {x → wartość}                                  jw.;
wyrażenie /. {x → wart1, y → wart2}          wykonanie dwóch podstawień jednocześnie.
```

W celu wstawienia strzałki możemy użyć, znanego z wcześniejszych wersji programu, zapisu znak minus i znak relacji większości (- >) lub przycisku z palety *Basic Math Input* (→). Oznaczmy przez v wielomian trzeciego stopnia.

```
In[49]:= v = x^3 - x + 1

Out[49]= 1 - x + x^3
```

Policzmy jego wartość w punkcie x=2.

```
In[50]:= v /. x → 2

Out[50]= 7
```

Możemy także regułę podstawienia umieścić w nawiasach klamrowych.

```
In[51]:= v /. {x → 5}

Out[51]= 121
```

Dokonajmy teraz podstawienia postaci x=1-y.

```
In[52]:= v /. x → 1 - y

Out[52]= (1 - y)^3 + y
```

Nadajmy zmiennej x wartość liczbową.

```
In[53]:= x = 4
Out[53]= 4
```

Zobaczmy jakie jest teraz v.

```
In[54]:= v
Out[54]= 61
```

Anulujmy definicję zmiennej x i ponownie wydrukujmy zmienną v.

```
In[55]:= x =.; v
Out[55]= 1 - x + x^3
```

Podstawmy wartości dwóch zmiennych jednocześnie.

```
In[56]:= x^2 + y^2 - 4 /. {x → 2, y → a}
Out[56]= a^2
```

Możemy definiować zmienną także następująco:

```
x := wzór definiowanie zmiennej.
```

Jest to tzw. opóźnione przypisanie. Różnica pomiędzy oboma sposobami określania zmiennej x polega na tym, że przy definicji x=... *Mathematica* wykorzystuje dostępne wartości zmiennych i pod zmienną x zapisywany jest ostateczny wynik obliczeń. W przypadku definicji x:=... *Mathematica* zapamiętuje pod zmienną cały podany wzór nie przekształcając go. Zilustrujmy, omawiane różnice przykładem.

```
In[57]:= Clear[x, y, a, b]
In[58]:= a = 2; b = 3;
In[59]:= x = a + b
Out[59]= 5
In[60]:= y := a + b
In[61]:= y
Out[61]= 5
In[62]:= {x, y}
Out[62]= {5, 5}
In[63]:= b = 4
Out[63]= 4
In[64]:= {x, y}
Out[64]= {5, 6}
```

W pakiecie Mathematica dostępne jest także tzw. opóźnione podstawienie:

```
     wyrażenie /. x :→ wartość
    wyrażenie /. {x :→ wartość}
wyrażenie /. {x :→ wart1, y :→ wart2}
```

Symbol opóźnionego podstawienia (:→) uzyskamy pisząc kolejno dwukropek i znak większości (:>). W podstawieniu wartość (prawa strona podstawienia) obliczana jest raz na początku i następnie wykorzystywana do wszystkich podstawień. Natomiast w przypadku opóźnionego podstawienia wartość jest wyznaczana na nowo przy każdym podstawieniu.

Najlepiej różnice zilustrować przykładem.

```
In[65]:= Clear[a, b];
         {a, b, a, b, a} /. {a → RandomInteger[{0, 100}]}
Out[66]= {100, b, 100, b, 100}
```

W przypadku zwykłego podstawienia najpierw została wygenerowana całkowita liczba pseudolosowa, a następnie podstawiona w każde miejsce wystąpienia symbolu a.

```
In[67]:= {a, b, a, b, a} /. {a :→ RandomInteger[{0, 100}]}
Out[67]= {99, b, 36, b, 90}
```

Z kolei w przypadku opóźnionego podstawienia do każdego zastąpienia symbolu a była generowana nowa liczba.

Opisane powyżej przypisania i podstawienia możemy równoważnie zapisać, korzystając z odpowiedniej instrukcji.

```
 x = wartość        równoważna postać Set[x, wartość];
x := wartość   równoważna postać SetDelayed[x, wartość];
 x → wartość       równoważna postać Rule[x, wartość];
x :→ wartość  równoważna postać RuleDelayed[x, wartość].
```

Komórki wejściowe zaczynają się od etykiety In[n]:=, natomiast wynikowe etykietą Out[n]=. Oznacza to, że polecenie wejściowe jest zapisane pod symbolem In[n], a odpowiadający mu wynik pod symbolem Out[n]. Dzięki temu można je łatwo wykorzystać w dalszych obliczeniach. Symbol := przy etykiecie wejściowej wskazuje na „opóźnienie", czyli wykorzystanie In[n] spowoduje ponowne przeliczenie polecenia wejściowego. W komórce wynikowej nie mamy „opóźnienia", czyli wynik zostanie przepisany bez ponownych obliczeń.

Zobaczmy, jak to wygląda w praktyce.

```
In[68]:= Clear[a, b];
         a = 10;
In[70]:= a + b + 21
Out[70]= 31 + b
```

Zmieńmy teraz wartość wykorzystywanej zmiennej.

```
In[71]:= a = 1;
```

Przeliczmy jeszcze raz wcześniejsze polecenie wejściowe:

```
In[72]:= In[70]
Out[72]= 22 + b
```

oraz wykorzystajmy w obliczeniach wcześniejszy wynik:

```
In[73]:= Out[70] * 7 // Expand
Out[73]= 217 + 7 b
```

1.9. Porównania i podstawienia

```
In[74]:= Clear["Global`*"]
```

Musimy rozróżniać zapisy x=y, x= =y i x= = =y. Poniższe zestawienie wyjaśni nam, czym się one różnią.

```
 x = y  podstawienie;
 x == y    równość;
x === y identyczność.
```

Przypiszmy x wartość 6.

```
In[75]:= x = 6
Out[75]= 6
```

Sprawdźmy, czy x jest równe 6.

```
In[76]:= x == 6
Out[76]= True
```

W przypadku operacji = = = otrzymamy ten sam wynik.

```
In[77]:= x === 6
Out[77]= True
```

Operacja == nie może podać konkretnego wyniku, jeżeli wartość liczbowa zmiennej jest nieznana.

```
In[78]:= t == 4
Out[78]= t == 4
```

Zobaczmy, jaki wynik da porównanie przy pomocy trzech znaków równości (= = =).

```
In[79]:= t === 4
Out[79]= False
```

Uwidoczniła się tu różnica między porównaniami = = i = = =. Pierwsze z nich (= =) daje w wyniku True lub False tylko wtedy, gdy wyrażenia mają konkretną wartość liczbową lub identyczną postać. Stąd też właśnie porównanie = = używane jest do zapisu równań. Natomiast drugie z omawianych porównań (= = =) sprawdza, czy wyrażenia są identyczne (bez względu jakie wartości liczbowe by się do nich podstawiło). Jeśli tak jest, to daje ono w wyniku True, jeśli nie, to False.

2. Elementy programowania

2.1. Pętle

```
In[80]:= Clear["Global`*"]
```

Wykonując obliczenia często spotykamy się z zadaniem wielokrotnego powtarzania takiej samej czynności. Jest tak na przykład przy sumowaniu wyrazów ciągu. Rozwiązanie polegające na „ręcznym" wykonaniu takiej czynności tyle razy, ile trzeba, często nie wchodzi w rachubę ze względów praktycznych. Wyobraźmy sobie, że mamy zesumować 10 milionów wyrazów ciągu. Do takich celów używamy pętli. Generalnie rozróżniamy pętle o określonej liczbie powtórzeń i o nieokreślonej liczbie powtórzeń. Ten ostatni przypadek ma miejsce wtedy, gdy koniec obliczeń zależy od ich wyniku, a nie od liczby powtórzeń. *Mathematica* dysponuje trzema instrukcjami pętli: o określonej liczbie powtórzeń (pętla Do), o nieokreślonej liczbie powtórzeń (pętla While), oraz instrukcją For, która może zrealizować pętle obu typów.

```
Do[treść, {licznik, start, stop, krok}]        cyklicznie wykonuje treść;
                                          licznik na początku przybiera wartość start
                                          i kolejno powiększa się o wartość krok dotąd,
                                              dokąd nie przewyższy wartości stop;

   For[start, warunek, krok, treść]      pętla wykonywana dopóki warunek jest spełniony,
                                                  poczynając od startu;

        While[warunek, treść]                  treść jest wykonywana tak długo,
                                          jak długo warunek ma wartość logiczną prawda (True);
                                          wartość logiczna warunku sprawdzana jest każdorazowo
                                                  przed wykonaniem treści.
```

Wypisać kolejne liczby naturalne od 1 do 5.
Mamy tu do czynienia z sytuacją, w której pięć razy wykonamy czynność wypisania. Należy więc skorzystać z pętli Do.

```
Print[wyrażenie1, ...]  wypisuje podane wyrażenia.
```

Rozwiązanie zadania.

```
In[81]:= Do[Print[i], {i, 1, 5}];
```

```
1
2
3
4
5
```

Policzyć sumę czwartych potęg liczb naturalnych od 1 do 100.

Ponieważ będziemy sumowali liczby, musimy gdzieś przechowywać sumy cząstkowe. Użyjemy do tego zmiennej `s`.

```
In[82]:= Clear[s];
         s = 0;
         Do[s += i^4, {i, 1, 100}];
         s
Out[85]= 2 050 333 330
```

Wyznaczyć maksymalną liczbę naturalną n o tej własności, że suma czwartych potęg wszystkich liczb naturalnych nie większych od n jest mniejsza od miliona.

Zauważmy, że użycie pętli `Do` jest w tym przypadku niezbyt wygodne: ponieważ nie wiemy, ile liczb dodać, musielibyśmy ustalić wynik na drodze eksperymentalnej. Gdyby ograniczeniem nie był milion, lecz jakaś bardzo duża liczba, ustalenie wyniku metodą prób i błędów mogłoby okazać się niemożliwe (np. ze względu na czas obliczeń). Wykorzystamy więc pętlę `While`. Podobnie jak poprzednio sumy cząstkowe będziemy przechowywać w zmiennej `s`, zmienna `i` posłuży nam natomiast do zapisu kolejno sumowanych liczb.

```
In[86]:= Clear[i, s, n];
         i = 0;
         s = 0;
         While[s ≤ 1 000 000, i++; s += i^4];
         n = i - 1
Out[90]= 21

In[91]:= s - i^4
Out[91]= 917 147

In[92]:= i
Out[92]= 22

In[93]:= s
Out[93]= 1 151 403
```

Liczba `n` jest o 1 mniejsza od ostatniej wartości licznika `i`, gdyż po ostatniej mniejszej bądź równej milionowi wartości `s` nastąpi jeszcze raz zwiększenie licznika o jeden.

Wypisać te silnie liczb naturalnych, które są mniejsze od tysiąca.

Wykorzystamy teraz pętlę `For`.

```
In[94]:= For[i = 0, i! < 10^3, i++, Print[i, "!= ", i!]]
```

```
0! = 1

1! = 1

2! = 2

3! = 6

4! = 24

5! = 120

6! = 720
```

2.2. Instrukcje warunkowe

Pozwalają one modyfikować bieg programu w zależności od wyników przeprowadzonych obliczeń, wprowadzonych danych itp. W programie *Mathematica* są dostępne trzy tego typu instrukcje. Zaczniemy od omówienia instrukcji If.

Instrukcja If

In[95]:=
```
Clear["Global`*"]
```

```
If[warunek, pinstr, finstr, ninstr]          sprawdza, jaką wartość logiczną ma warunek;
                                        jeśli warunek == True (prawda), wykonywana jest instrukcja pinstr,
                                         jeśli warunek == False (fałsz) wykonywana jest instrukcja finstr,
                                   jeśli warunek nie ma określonej wartości logicznej, wykonywana jest instrukcja ninstr;

   If[warunek, pinstr, finstr]          jeśli warunek == True (prawda), wykonywana jest instrukcja pinstr,
                                         jeśli warunek == False (fałsz) wykonywana jest instrukcja finstr;

      If[warunek, pinstr]               jeśli warunek == True (prawda), wykonywana jest instrukcja pinstr.
```

Instrukcję If przedstawimy, korzystając z prostych przykładów.

Porównywanie dwóch liczb; $a \geq b$:

In[96]:=
```
Clear[a, b];
a = 1; b = 0;
```

In[98]:=
```
If[a < b, Print["a<b"], Print["a≥b"], Print["Nie da się nic powiedzieć"]]
```

```
a≥b
```

Porównywanie dwóch liczb; $a < b$:

In[99]:=
```
Clear[a, b];
a = 1; b = 2;
```

In[101]:=
```
If[a < b, Print["a<b"], Print["a≥b"], Print["Nie da się nic powiedzieć"]]
```

```
a<b
```

Porównanie dwóch liczb; b nieokreślone:

In[102]:=
```
Clear[a, b];
a = 1;
```

In[104]:=
```
If[a < b, Print["a<b"], Print["a≥b"], Print["Nie da się nic powiedzieć"]]
```

```
Nie da się nic powiedzieć
```

Instrukcja Switch

In[105]:=
```
Clear["Global`*"]
```

```
Switch[wyrażenie, forma₁, wartość₁, ...] oblicza wyrażenie i porównuje jego wartość z formaᵢ;
                                          zwraca wartośćᵢ dla pierwszego spełnionego warunku
                                                wyrażenie == formaᵢ.
```

Przykłady.

In[106]:=
```
a = 1; b = 2;
```

In[107]:=
```
Clear[x, y];
```

In[108]:=
```
Switch[a + b, 2, x, 3, y, 1, x + y]
```

Out[108]=
```
y
```

In[109]:=
```
Switch[b – a, 2, x, 3, y, 1, x + y]
```

Out[109]=
```
x + y
```

In[110]:=
```
Switch[2 a + b, 2, x, 3, y, 1, x + y]
```

Out[110]=
```
Switch[4, 2, x, 3, y, 1, x + y]
```

In[111]:=
```
Switch[2 a + b, 2, x, 3, y, 1, x + y, _, 7]
```

Out[111]=
```
7
```

Instrukcja Which

In[112]:=
```
Clear["Global`*"]
```

```
Which[test₁, wartość₁, test₂, wartość₂, ...] oblicza testᵢ zwracając pierwszą wartośćᵢ,
                                              dla której testᵢ był równy True.
```

Przykłady.

In[113]:=
```
a = 1; b = 2;
```

In[114]:=
```
Which[a == b, x, a < b, y, a ≤ b, x + y]
```

Out[114]=
```
y
```

In[115]:=
```
Which[a == b, x, a < b – 1, y, a > b, x + y]
```

In[116]:=
```
Which[a == b, x, a < b – 1, y, a > b, x + y, True, x * y]
```

Out[116]=
```
x y
```

2.3. Instrukcje Module i Block

In[117]:=
```
Clear["Global`*"]
```

Przedstawimy teraz instrukcje wykorzystywane do pisania funkcji i procedur.

```
      Module[{u, v, ...}, treść]        wykonuje treść, używając zmiennych lokalnych u, v, ...;

Module[{u, v, ..., t = t₀, ...}, treść]          jak wyżej; zmiennej lokalnej t zostaje
                                                      nadana wartość początkowa t₀;

      Block[{u, v, ...}, treść]       wykonuje treść, używając zmiennych globalnych u, v, ... ,
                                                  którym nadawane są wartości lokalne;
                                          ich poprzednie wartości przechowywane są na stosie
                                                  i odtwarzane po wyjściu z procedury;
```

```
Block[{u, v, ..., t = t0, ...}, treść]  jak wyżej; zmiennej t zostaje nadana wartość początkowa t0.
```

Instrukcja Module pozwala wprowadzać do obliczeń zmienne lokalne. Zmienne te są odróżniane od zmiennych o tych samych nazwach występujących poza procedurą, poprzez dodanie na końcu nazwy znaku $ oraz pewnego numeru, który jest identyfikatorem bloku w systemie, ukrytym m.in. w zmiennej $ModuleNumber. Takie przyporządkowanie liczb kolejnym blokom pozwala uniknąć kolizji nazw zmiennych np. w sytuacji, gdy wewnątrz funkcji wywoływana jest ta sama funkcja, ponieważ każdy nowo powstały blok ma inny numer.

Zdefiniujmy trzy zmienne, a następnie wykorzystajmy instrukcję Module.

```
In[118]:= a = 1; b = 2; c = 3;
In[119]:= Module[{a = 2, b = 1, d}, d = a^b; c = d; d]
Out[119]= 2
```

Sprawdźmy, jakie wartości mają zdefiniowane wcześniej zmienne.

```
In[120]:= {a, b, c}
Out[120]= {1, 2, 2}
```

Zmienne a i b zostały zdefiniowane w Module jako zmienne lokalne, dlatego ich wartości na zewnątrz nie uległy zmianie. Natomiast zmienna c była wykorzystywana jako zmienna globalna, więc jej wartość uległa zmianie.

Innym rozwiązaniem zapobiegającym kolizji symboli o tych samych nazwach jest zapamiętywanie np. na stosie, starej wartości zmiennej przy wejściu do bloku i odtworzenie tej wartości przy wyjściu. Tak zachowują się w programie *Mathematica* m.in. bloki o nazwie Block.

```
In[121]:= a = 1; b = 2; c = 3;
In[122]:= Block[{a = 2, b = 1, d}, d = a - 5 b; c = d; d]
Out[122]= -3
```

Sprawdźmy, jakie wartości mają zdefiniowane wcześniej zmienne.

```
In[123]:= {a, b, c}
Out[123]= {1, 2, -3}
```

Różnicę w działaniu instrukcji Block i Module rozpatrzymy na następującym przykładzie.

```
In[124]:= Clear[a, b, c];
          a = Sin[b]
Out[125]= Sin[b]
In[126]:= Module[{b = c}, b + a]
Out[126]= c + Sin[b]
In[127]:= Block[{b = c}, b + a]
Out[127]= c + Sin[c]
```

Block wykorzystuje więc również powiązania pomiędzy zmiennymi określone na zewnątrz.

Zobaczmy jeszcze jeden przykład ilustrujący różnice między obu instrukcjami. Napiszemy (wybiegając trochę do przodu) procedury liczące sumę:

$$\sum_{i=1}^{10} x^i$$

```
In[128]:= Clear[p1];
          p1[x_] := Module[{i, s = 0},
            Do[s += x^i, {i, 1, 10}];
            Return[s]]

In[130]:= Clear[p2];
          p2[x_] := Block[{i, s = 0},
            Do[s += x^i, {i, 1, 10}];
            Return[s]]
```

Obie procedury zwrócą ten sam wynik, gdy jako argumentu użyjemy wartości liczbowej:

```
In[132]:= {p1[2], p2[2]}

Out[132]= {2046, 2046}
```

a także dla większości wartości symbolicznych.

```
In[133]:= {p1[t], p2[t]}

Out[133]= {t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + t^9 + t^10, t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + t^9 + t^10}
```

Problem pojawi się dopiero wtedy, gdy jako wartości symbolicznej użyjemy nazwy zgodnej z którąś ze zmiennych lokalnych (np. `i`):

```
In[134]:= Clear[i];
          {p1[i], p2[i]}

Out[135]= {i + i^2 + i^3 + i^4 + i^5 + i^6 + i^7 + i^8 + i^9 + i^10, 10 405 071 317}
```

W instrukcji `Block` zmienne globalne o tych samych nazwach co lokalne są zapamiętywane, a nazwy ich lokalnych odpowiedników pozostają bez zmian. W naszym przypadku zmienną globalną jest zmienna `i`, ale nie ma żadnej wartości, więc nie ma co zapamiętywać. Później wszystkie znajdujące się w definicji wystąpienia zmiennej `x` są zastępowane aktualnym argumentem, czyli zmienną `i`. Dlatego funkcja ta oblicza w zasadzie wartość następującej sumy:

$$\sum_{i=1}^{10} i^i$$

Sprawdźmy to:

```
In[136]:= Sum[i^i, {i, 1, 10}]

Out[136]= 10 405 071 317
```

Jeszcze jeden przykład, tym razem z obliczeniami rekurencyjnymi:

```
In[137]:= cl[1] = 1;
          cl[n_Integer?EvenQ] := cl[n / 2] + 1
          cl[n_Integer?OddQ] := cl[3 n + 1] + 1
```

Usuńmy ograniczenie na zakres rekurencji i policzmy wartość cl[9 780 657 630]:

```
In[140]:= Block[{$RecursionLimit = Infinity}, cl[9 780 657 630]]

Out[140]= 1133
```

Ustawienia domyślne nie pozwalają policzyć tej wartości:

```
In[141]:= cl[9 780 657 630]

          $RecursionLimit: Recursion depth of 1024 exceeded.

Out[141]= 1 + TerminatedEvaluation[RecursionLimit]
```

2.4. Definiowanie funkcji i procedur

In[142]:=
```
Clear["Global`*"]
```

Oprócz funkcji zdefiniowanych już przez twórców programu, możemy używać własnych, zdefiniowanych przez siebie.

```
nazwaFunkcji[arg1_, ...] := wyrażenie          definiowanie funkcji;
              Clear[f]                  odwołanie wszystkich definicji dla f;
            ClearAll[f]                     odwołanie definicji funkcji f
                                        i usunięcie przypisanych jej atrybutów,
                                      gdy funkcja nie posiada atrybutu Protected;
                ?f                            wypisanie definicji f.
```

A zatem: podajemy nazwę funkcji, listę jej argumentów umieszczoną w nawiasach prostokątnych, dwuznak „:=" i po nim - definicję funkcji. Nazwa funkcji oraz nazwy argumentów nie mogą zawierać znaku odstępu, znaku podkreślenia i znaków specjalnych oraz muszą zaczynać się literą. Proponujemy stosowanie zasady zalecanej przez twórców programu: nazwy definiowanych przez nas obiektów zaczynamy małą literą (pierwsza duża litera oznaczać wtedy będzie obiekt zdefiniowany przez autorów programu). Jeśli nazwa składa się z wielu wyrazów, każdy poza pierwszym rozpoczynamy dużą literą. Użycie dużych liter poprawi czytelność nazw. Nazwy argumentów po lewej stronie definicji kończymy znakiem podkreślenia. Po zdefiniowaniu funkcji możemy używać ich tak, jak i zwykłych instrukcji programu *Mathematica*.
W definicji funkcji lub procedury przy nazwach ich parametrów formalnych, po lewej stronie definicji, dodajemy znak podkreślenia. Znaku tego nie dodajemy po prawej stronie równości definiującej.
Funkcje i procedury wywołujemy, podając ich nazwy oraz zastępując parametry formalne parametrami aktualnymi, które mogą być w ogólności również wyrażeniami. Parametr aktualny odpowiada temu parametrowi formalnemu, który zajmuje to samo miejsce na liście parametrów funkcji lub procedury.

Uwaga. Mathematica odróżnia duże i małe litery!

Definicje dwóch różnych funkcji:

In[143]:=
```
Clear[fn]; fn[x_] := 2 x + 1
```

In[144]:=
```
Clear[fN]; fN[x_] := x + 5
```

Policzenie ich wartości w punkcie x=2:

In[145]:=
```
fn[2]
```

Out[145]=
```
5
```

In[146]:=
```
fN[2]
```

Out[146]=
```
7
```

Możemy także liczyć wartości funkcji dla argumentów będących symbolami.

In[147]:=
```
fn[t]
```

Out[147]=
```
1 + 2 t
```

Sprawdźmy, jak jest zdefiniowana funkcja fN.

In[148]:=
```
?fN
```

Out[148]=

Symbol

Global`fN

Definitions

fN[x_] := x + 5

Full Name Global`fN

Możliwe jest także definiowanie funkcji, których wartość zależy od pewnych warunków nakładanych na argumenty. Wszystkie warunki podajemy po wyrażeniu, zaczynać się one powinny od znaków /;. Z ich pomocą można łatwo zdefiniować takie funkcje jak np. |x|.

```
nazwaFunkcji[arg1_, ...] := wyrażenie1 /; warunek1
nazwaFunkcji[arg1_, ...] := wyrażenie2 /; warunek2
                          ⋮                          definiowanie warunkowe funkcji.
nazwaFunkcji[arg1_, ...] := wyrażenien /; warunekn
nazwaFunkcji[arg1_, ...] := wyrażenie1 /; warunek1
```

Zdefiniujmy funkcję f:

In[149]:=
```
Clear[f];
f[x_] := x + 3;
f[x_] := x - 1 /; x < 1;
f[x_] := x^2 /; 0 ≤ x ≤ 2;
f[-2] := -10;
```

W powyższej definicji wiele dziedzin zachodzi wzajemnie na siebie. Prześledźmy, jak *Mathematica* rozstrzygnęła te sprzeczności - wystarczy obliczyć wartości funkcji f dla kilku charakterystycznych punktów:

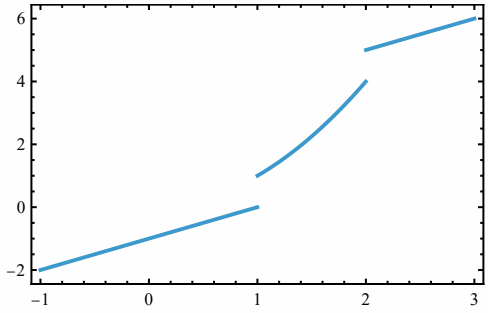
In[154]:=
```
{f[-3], f[-2], f[-1], f[0], f[1], f[2], f[5]}
```

Out[154]=
```
{-4, -10, -2, -1, 1, 4, 8}
```

oraz wykreślmy jej wykres.

In[155]:=
```
Plot[f[x], {x, -1, 3}, Frame → True, Axes → None, Exclusions → {1, 2}, ImageSize → 250]
```

Out[155]=



In[156]:=
```
?f
```

Out[156]=

Symbol

Global`f

Definitions

```
f[-2] := -10

f[x_] := x - 1 /; x < 1

f[x_] := x^2 /; 0 ≤ x ≤ 2

f[x_] := x + 3
```

Full Name Global`f

Zmieńmy kolejność warunków:

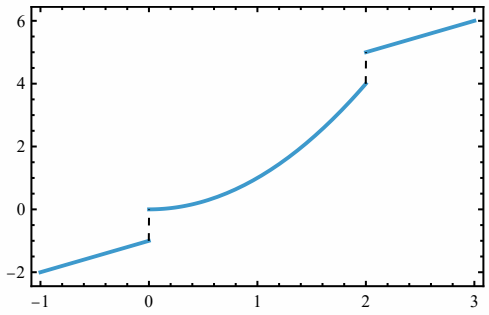
In[157]:=
```
Clear[f1];
f1[x_] := x + 3;
f1[x_] := x^2 /; 0 ≤ x ≤ 2;
f1[x_] := x - 1 /; x < 1;
f1[-2] := -10;
```

In[162]:=
```
{f1[-3], f1[-2], f1[-1], f1[0], f1[1], f1[2], f1[5]}
```

Out[162]=
```
{-4, -10, -2, 0, 1, 4, 8}
```

In[163]:=
```
Plot[f1[x], {x, -1, 3}, Frame → True, Axes → None,
 Exclusions → {0, 2}, ExclusionsStyle → Dashed, ImageSize → 250]
```

Out[163]=



In[164]:=
```
?f1
```

Out[164]=

Symbol

Global`f1

Definitions

```
f1[-2] := -10

f1[x_] := x^2 /; 0 ≤ x ≤ 2

f1[x_] := x - 1 /; x < 1

f1[x_] := x + 3
```

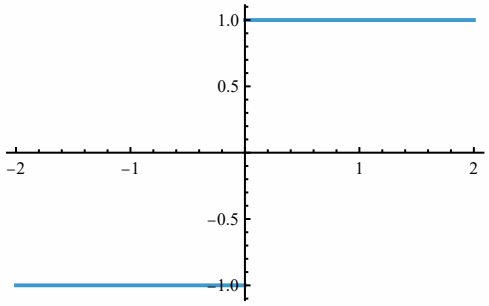
Full Name Global`f1

W definicji funkcji można także wykorzystać instrukcje warunkowe.

In[165]:=
```
Clear[fg];
fg[x_] := If[x > 0, 1, -1]
```

In[167]:=
```
Plot[fg[x], {x, -2, 2}, Exclusions → {0}, ImageSize → 250]
```

Out[167]=



```
In[168]:= fg[0]
Out[168]= -1
```

Przykład definicji ciągu.

```
In[169]:= Clear[aa, n, np];
          aa[n_] := Which[EvenQ[n], p, OddQ[n], np]
In[171]:= Table[aa[ns], {ns, 1, 10}]
Out[171]= {np, p, np, p, np, p, np, p, np, p}
```

Jeszcze jeden przykład.

```
In[172]:= Clear[ff];
          ff[x_] := Which[x < -1, -1, x == -1, 2, x ≤ 2, 1, True, 3]
In[174]:= {ff[-2], ff[-1], ff[0], ff[4]}
Out[174]= {-1, 2, 1, 3}
```

Procedury możemy definiować następująco:

```
nazwaProcedury[arg1_, ...] := Module[{zmienne lokalne}, treść]      zdefiniowanie procedury;
    nazwaProcedury[arg1_, ...] := Block[{zmienne}, treść]           zdefiniowanie procedury;
        nazwaProcedury[arg1_, ...] := (treść)                       zdefiniowanie procedury;

                    Return[x]                       zamyka procedurę i zwraca wartość zmiennej x;
                 Return[{x, y}]                      zamyka procedurę i zwraca dwie wartości;
                   Return[]                    zamyka procedurę, nie zwracając żadnej wartości.
```

Napiszemy procedurę, której argumentami będą x i n, a wynikiem której będzie suma:

$$\sum_{i=0}^{n} \frac{x^i}{i!}$$

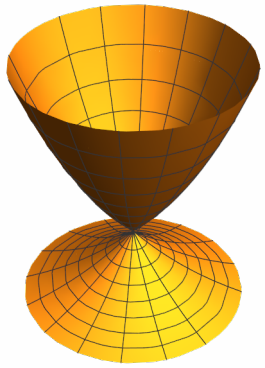
```
In[175]:= Clear[edox];
          edox[x_, n_] := Module[{i, s = 1},
            Do[s += x^i/i!, {i, 1, n}];
            Return[s]]
In[177]:= edox[1, 5]
Out[177]= 163/60
In[178]:= edox[x, 4]
Out[178]= 1 + x + x^2/2 + x^3/6 + x^4/24
In[179]:= Clear[a, b, u, v]
```

Możemy także dzielić argumenty na grupy i umieszczać w osobnych nawiasach kwadratowych.

```
In[180]:= pucharC[a_, b_][u_, v_] := {a * u * Cos[v], a * u * Sin[v], b * Exp[u]}
```

```
In[181]:= ParametricPlot3D[Evaluate[pucharC[1, 1][u, v]],
           {u, -1, 1}, {v, -π, π}, Axes → False, Boxed → False, ImageSize → 200]
```

Out[181]=



Możemy także definiować ciągi (funkcje) zadane rekurencyjnie.

```
In[182]:= Clear[a];
          a[1] = 1;
          a[2] = 1;
          a[n_] := a[n - 1] + a[n - 2]

In[186]:= Timing[Table[a[n], {n, 1, 30}]]

Out[186]= {2.90625, {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181,
            6765, 10 946, 17 711, 28 657, 46 368, 75 025, 121 393, 196 418, 317 811, 514 229, 832 040}}

In[187]:= Clear[b];
          b[1] = 1;
          b[2] = 1;
          b[n_] := b[n] = b[n - 1] + b[n - 2]

In[191]:= Timing[Table[b[n], {n, 1, 30}]]

Out[191]= {0., {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181,
            6765, 10 946, 17 711, 28 657, 46 368, 75 025, 121 393, 196 418, 317 811, 514 229, 832 040}}
```

Instrukcja Timing podaje czas wykonania instrukcji. Drugi zapis przyspiesza obliczenia wyrażeń rekurencyjnych. Przy tym zapisie, jeśli raz policzymy wartość b[n], to zostanie ona zapamiętana i będzie wykorzystywana przy kolejnym odwołaniu. Przy pierwszym zapisie wartość a[n] jest liczona przy każdym odwołaniu na nowo.

2.5. Usuwanie definicji

```
In[192]:= Remove["Global`*"]
```

Do usuwania definicji pojedynczych symboli możemy wykorzystać instrukcję Clear. Możemy także od razu usunąć definicje wszystkich zdefiniowanych przez nas symboli, co jest bardzo przydatne w sytuacji gdy zakończyliśmy jedno zadanie i chcemy przejść do następnego.

```
  Clear[symbol1, ...]                    usuwa definicje podanych symboli;
Clear[''kontekst`@'']            usuwa definicje wszystkich symboli zdefiniowanych
                              w zadanym środowisku, których nazwy zapisane są małymi literami;
Clear[''kontekst`*''] usuwa definicje wszystkich symboli zdefiniowanych w zadanym środowisku;
Remove[''kontekst`@'']           działa podobnie jak odpowiednia instrukcja Clear,
                              usuwając jednocześnie informację o wykorzystaniu obiektu;
Remove[''kontekst`*'']           działa podobnie jak odpowiednia instrukcja Clear,
                              usuwając jednocześnie informację o wykorzystaniu obiektu.
```

Mathematica pozwala na tworzenie odrębnych środowisk dla poszczególnych fragmentów projektu programistycznego (tzw. kontekstów). Ma to zastosowanie m.in. do podziału zadania na fragmenty, a także wtedy, gdy chcemy na chwilę uruchomić inne zadanie, a potem wrócić do głównego środowiska (np. w celu otrzymania potrzebnych wyników). Definiowanie nowych symboli grozi mody-

fikacją już istniejących, w przypadku zgodnych nazw. Utworzenie nowego środowiska zwalnia nas od zastanawiania się, czy dana nazwa została już użyta, bowiem powrót do starego kontekstu spowoduje odtworzenie wszystkich poprzednich definicji. Nazwa aktualnego kontekstu ukryta jest w zmiennej systemowej $Context.

Zapisy postaci Clear["`@"],... powodują usuwanie odpowiednich definicji w aktualnym środowisku.

Sprawdźmy, jak nazywa się aktualne środowisko.

In[193]:=
```
$Context
```

Out[193]=
```
Global`
```

Zdefiniujmy kilka symboli.

In[194]:=
```
x = 6;
f1[x_] := x^2;
y = x + 2;
z := x - 2;
F2[s_] := Sin[s];
```

Sprawdźmy ich nagłówki.

In[199]:=
```
{Head[x], Head[y], Head[z], Head[f1], Head[F2]}
```

Out[199]=
```
{Integer, Integer, Integer, Symbol, Symbol}
```

Zobaczmy także jak działają.

In[200]:=
```
{x, y, z, f1[t], F2[t]}
```

Out[200]=
$$\{6, 8, 4, t^2, \text{Sin}[t]\}$$

Sprawdźmy, jakie symbole, których nazwy zapisane są małymi literami, są zdefiniowane w głównym środowisku.

In[201]:=
```
?"Global`@"
```

Out[201]=
```
Global`
f1          s          t          x          y          z
```

Wszystkie symbole.

In[202]:=
```
?"Global`*"
```

Out[202]=
```
Global`
f1          F2          s          t          x          y          z
```

Usuńmy definicje symboli, których nazwy zapisane są małymi literami.

In[203]:=
```
Clear["Global`@"]
```

In[204]:=
```
{Head[x], Head[y], Head[z], Head[f1], Head[F2]}
```

Out[204]=
```
{Symbol, Symbol, Symbol, Symbol, Symbol}
```

Definicja funkcji F2 nie została usunięta.

In[205]:=
```
{x, y, z, f1[t], F2[t]}
```

Out[205]=
```
{x, y, z, f1[t], Sin[t]}
```

Usuńmy ją.

In[206]:=
```
Clear["Global`*"]
```

In[207]:=
```
F2[t]
```

Out[207]=
```
F2[t]
```

Zobaczmy, jakie symbole, których nazwy zapisane są małymi literami, występują lub były używane w głównym środowisku.

In[208]:=
```
?"Global`@"
```

Out[208]=
```
Global`
f1          s          t          x          y          z
```

A teraz wszystkie symbole.

In[209]:=
```
?"Global`*"
```

Out[209]=
```
Global`
f1          F2          s          t          x          y          z
```

Usuńmy informacje o używanych symbolach, których nazwy zapisane są małymi literami.

In[210]:=
```
Remove["Global`@"]
```

In[211]:=
```
?"Global`@"
```

Out[211]=
```
Missing[UnknownSymbol, Global`@]
```

Informacja o F2 pozostała.

In[212]:=
```
?"Global`*"
```

Out[212]=

Symbol

Global`F2

Full Name Global`F2

Usuńmy także ją.

In[213]:=
```
Remove["Global`*"]
```

In[214]:=
```
?"Global`*"
```

Out[214]=
```
Missing[UnknownSymbol, Global`*]
```

2.6. Argumenty

In[215]:=
```
Clear["Global`*"]
```

Argumentom funkcji możemy nadawać wartości domyślne.

```
arg_ : wartość    nadanie argumentowi wartości domyślnej.
```

Zobaczmy przykład.

In[216]:=
```
f[x_, y_ : 5] := x + y
```

In[217]:=
```
f[2, 3]
```

Out[217]=
```
5
```

In[218]:=
```
f[3]
```

Out[218]= 8

Kolejny przykład.

In[219]:=
```
g[x_ : 3, y_, z_ : 4] := x + y + z
```

Możemy wywoływać tę funkcję z trzema argumentami.

In[220]:=
```
g[1, 2, 3]
```

Out[220]= 6

Natomiast gdy wywołamy ją z dwoma argumentami, to zostanie wykorzystana wartość domyślna ostatniego argumentu, dla którego została ona określona.

In[221]:=
```
g[1, 1]
```

Out[221]= 6

Przy jednym argumencie w wywołaniu zostaną wykorzystane obie wartości domyślne.

In[222]:=
```
g[2]
```

Out[222]= 9

Można także definiować funkcje, które będą dopuszczały wywołanie z różną liczbą argumentów.

```
 arg_   dokładnie jeden argument;
arg__      jeden lub więcej;
arg___     zero lub więcej.
```

Często nadanie argumentowi wartości domyślnej może okazać się niewystarczającym sposobem rozwiązania problemu zmiennej liczby argumentów. Dzieje się tak np. w sytuacji, gdy nie znamy maksymalnej liczby argumentów bądź po prostu nie interesuje nas większość z nich, a nie chcemy by była im przypisywana jakaś wartość (np. w sortowaniu za każdym razem zamieniamy dwa argumenty, resztę argumentów pozostawiając niezmienioną). W takich sytuacjach może nam pomóc zastosowanie w miejsce jednego podkreślenia dwóch lub trzech podkreśleń. Różnica pomiędzy nimi jest taka, że podwójne podkreślenie stosujemy dla wskazania, iż w tym miejscu może wystąpić jedno lub więcej wyrażeń, natomiast potrójne podkreślenie dla zero lub więcej wyrażeń. Ilustruje to poniższy przykład.

In[223]:=
```
Clear[sortujr];
sortujr[{a___, x_, y_, b___}] := sortujr[{a, y, x, b}] /; x > y;
sortujr[w_] := w;
```

In[226]:=
```
sortujr[{3, 2, 6, 1, 0, 5, -1}]
```

Out[226]= {−1, 0, 1, 2, 3, 5, 6}

Kolejna instrukcja usunie sąsiednie powtarzające się elementy.

In[227]:=
```
Clear[usun];
usun[{a___, x_, x_, b___}] := usun[{a, x, b}];
usun[w_] := w;
```

In[230]:=
```
usun[{3, 5, 5, 5, 6, 1, 1, 7, 2, 2, 0, 0, 0}]
```

Out[230]= {3, 5, 6, 1, 7, 2, 0}

A teraz usunięcie wszystkich powtórzeń.

In[231]:=
```
Clear[usun2];
usun2[{a___, x_, c___, x_, b___}] := usun2[{a, x, c, b}];
usun2[w_] := w;
```

In[234]:=
```
usun2[{3, 5, 5, 5, 2, 1, 1, 7, 2, 2, 1, 3, 0, 7, 0}]
```

Out[234]= {3, 5, 2, 1, 7, 0}

Definiując funkcję możemy określać także typ argumentów dla jakich ma ona działać.

```
  arg_nagłówek              określa dopuszczalny nagłówek (typ) argumentu;
arg_?Instrukcja    dopuszcza te argumenty dla których podana Instrukcja daje wartość True.
```

Możemy wykorzystywać następujące instrukcje określające nagłówki:

```
 Integer   liczba całkowita;
Rational   liczba wymierna;
  Real    liczba rzeczywista;
 Complex   liczba zespolona;
  List         lista;
 Symbol       symbol.
```

oraz m.in. następujące instrukcje testujące argumenty:

```
  IntegerQ     liczba całkowita;
   EvenQ       liczba parzysta;
   OddQ       liczba nieparzysta;
  PrimeQ       liczba pierwsza;
 Positive      liczba dodatnia;
 Negative       liczba ujemna;
NonPositive   liczba niedodatnia;
NonNegative    liczba nieujemna;
  NumberQ    liczba dowolnego typu;
 NumericQ    wartość numeryczna;
  VectorQ          wektor;
  MatrixQ         macierz.
```

Zobaczmy, jak to działa na przykładach.

In[235]:=
```
Clear[f];
f[x_Real, n_Integer?Positive] := x^n
```

Funkcja będzie działała, gdy pierwszy argument będzie liczbą rzeczywistą, a drugi dodatnią liczbą całkowitą.

In[237]:=
```
f[2.4, 3]
```

Out[237]=
```
13.824
```

In[238]:=
```
f[1, 4]
```

Out[238]=
```
f[1, 4]
```

In[239]:=
```
f[1., 4]
```

Out[239]=
```
1.
```

In[240]:=
```
f[2.1, 4.1]
```

Out[240]=
```
f[2.1, 4.1]
```

In[241]:=
```
f[2.1, -1]
```

Out[241]=
```
f[2.1, -1]
```

Jeszcze jeden przykład.

In[242]:=
```
Clear[poch, t];
poch[f_, x_Symbol, n_Integer?NonNegative] := D[f, {x, n}]
```

In[244]:=
```
poch[y * t^2 + t^t, t, 2]
```

Out[244]=
```
t^(-1+t) + 2 y + t^t (1 + Log[t])^2
```

```
In[245]:= ff[x_] := x^2
In[246]:= poch[ff[x], x, 1]
Out[246]= 2 x
In[247]:= poch[ff[t], t, 1.]
Out[247]= poch[t^2, t, 1.]
In[248]:= x = 3
Out[248]= 3
In[249]:= poch[ff[x], x, 1]
Out[249]= poch[9, 3, 1]
```

Definicja ciągu określonego innymi wzorami dla wyrazów parzystych i nieparzystych.

```
In[250]:= Clear[aa];
          aa[1] = 1;
          aa[n_?EvenQ] := n / 2 + 1
          aa[n_?OddQ] := 3 n + 2
In[254]:= Table[aa[n], {n, 1, 20}]
Out[254]= {1, 2, 11, 3, 17, 4, 23, 5, 29, 6, 35, 7, 41, 8, 47, 9, 53, 10, 59, 11}
```

2.7. Opcje

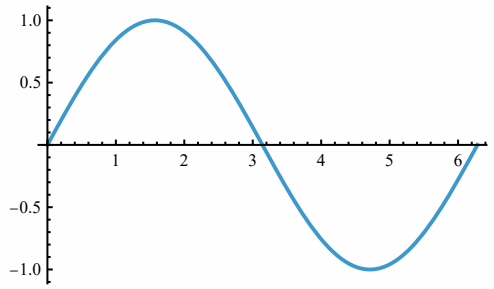
```
In[255]:= Clear["Global`*"]
```

Wiele ze standardowych instrukcji systemowych wyposażonych jest w pewien dodatkowy mechanizm warunkowego wykonywania, są to tzw. opcje. W działaniu opcje w zasadzie nie różnią się od argumentów posiadających wartość domyślną. Mają bowiem również pewną, z góry ustaloną wartość, którą użytkownik może, lecz nie musi zmienić.

```
instrukcja[argumenty]                wywołanie instrukcji z domyślnymi wartościami opcji;
instrukcja[argumenty, opcja → wartość]    wywołanie instrukcji ze zmianą wartości opcji.
```

Wywołajmy instrukcję Plot z domyślnymi wartościami wszystkich opcji.

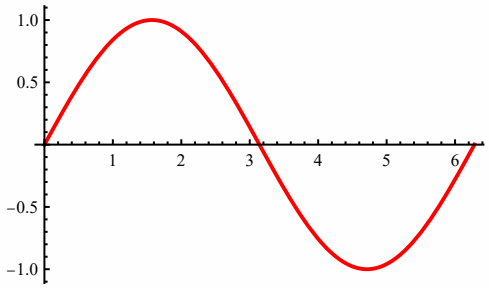
```
In[256]:= Plot[Sin[x], {x, 0, 2 π}, ImageSize → 250]
```



Zmieńmy wartość opcji PlotStyle.

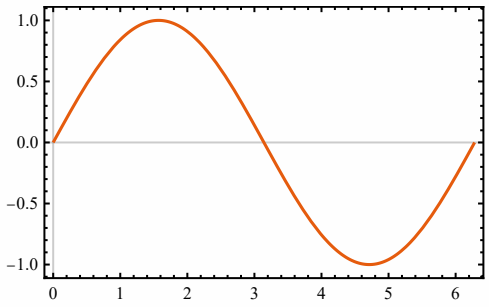
In[257]:=
```
Plot[Sin[x], {x, 0, 2 π}, PlotStyle → {Red}, ImageSize → 250]
```

Out[257]=



In[258]:=
```
Plot[Sin[x], {x, 0, 2 π}, PlotTheme → "Scientific", ImageSize → 250]
```

Out[258]=



Dostępne są instrukcje, które ułatwiają pracę z opcjami.

`Options[instrukcja]`	wyświetla listę opcji wraz z ich wartościami domyślnymi;
`Options[instrukcja, opcja]`	wyświetla wartość podanej opcji;
`SetOptions[instrukcja, opcja`$_1$` → wartość, ...]`	pozwala zmienić wartość domyślną opcji.

Zobaczmy, jakie opcja posiada instrukcja Plot.

In[259]:=
```
Options[Plot]
```

Out[259]=
```
{AlignmentPoint → Center, AspectRatio → 1/GoldenRatio, Axes → True, AxesLabel → None,
 AxesOrigin → Automatic, AxesStyle → {}, Background → None, BaselinePosition → Automatic,
 BaseStyle → {}, ClippingStyle → None, ColorFunction → Automatic, ColorFunctionScaling → True,
 ColorOutput → Automatic, ContentSelectable → Automatic, CoordinatesToolOptions → Automatic,
 DisplayFunction :→ $DisplayFunction, Epilog → {}, Evaluated → Automatic,
 EvaluationMonitor → None, Exclusions → Automatic, ExclusionsStyle → None, Filling → None,
 FillingStyle → Automatic, FormatType :→ TraditionalForm, Frame → False, FrameLabel → None,
 FrameStyle → {}, FrameTicks → Automatic, FrameTicksStyle → {}, GridLines → None,
 GridLinesStyle → {}, ImageMargins → 0., ImagePadding → All, ImageSize → Automatic,
 ImageSizeRaw → Automatic, LabelingSize → Automatic, LabelStyle → {}, MaxRecursion → Automatic,
 Mesh → None, MeshFunctions → {#1 &}, MeshShading → None, MeshStyle → Automatic,
 Method → Automatic, PerformanceGoal :→ $PerformanceGoal, PlotHighlighting → Automatic,
 PlotLabel → None, PlotLabels → None, PlotLayout → Automatic, PlotLegends → None,
 PlotPoints → Automatic, PlotRange → {Full, Automatic}, PlotRangeClipping → True,
 PlotRangePadding → Automatic, PlotRegion → Automatic, PlotStyle → Automatic,
 PlotTheme :→ $PlotTheme, PreserveImageOptions → Automatic, Prolog → {},
 RegionFunction → (True &), RotateLabel → True, ScalingFunctions → None, TargetUnits → Automatic,
 Ticks → Automatic, TicksStyle → {}, WorkingPrecision → MachinePrecision}
```

Zobaczmy wartość domyślną opcji PlotPoints.

In[260]:=
```
Options[Plot, PlotPoints]
```

Out[260]=
```
{PlotPoints → Automatic}
```

Pokażemy teraz, jak definiować opcje dla własnych funkcji.

```
   Options[f] = {nazwa → wartość}          definiuje opcję (nazwa) dla funkcji f;
Options[f] = {n1 → w1, n2 → w2, ...}       zdefiniowanie kilku opcji dla funkcji f;
SetOptions[f, n1 → w1, n2 → w2, ...]                    jw.;
     f[arg_, opcje___] := wzór      zdefiniowanie funkcji f dopuszczajacej wykorzystanie opcji;
 f[arg_, OptionsPattern[]] := wzór                      jw.;
     OptionValue[nazwaOpcji]          odczytanie wartości opcji, gdy w definicji funkcji
                                           wykorzystano instruksję OptionsPattern;
nazwaOpcji /. {opcje} /. Options[f]   odczytanie wartości opcji zadanej przez użytkownika
                                              lub wartości domyślnej.
```

Napiszemy prostą procedurę wykorzystującą opcje.

In[261]:=
```
Clear[ff];
Options[ff] = {a → 5, b → 1};
ff[x_, OptionsPattern[]] := Module[{c, a1, b1},
  a1 = OptionValue[a];
  b1 = OptionValue[b];
  c = x + a1 + b1;
  Return[c]]
```

In[264]:=
```
ff[2]
```

Out[264]=
```
8
```

In[265]:=
```
ff[2, a → 7]
```

Out[265]=
```
10
```

In[266]:=
```
ff[2, a → 7, b → 6]
```

Out[266]=
```
15
```

Napiszemy procedurę, która w zależności od wartości opcji będzie sortowała rosnąco lub malejąco. W tym celu napiszemy najpierw dwie procedury pomocnicze sortujące rosnąco lub malejąco.

In[267]:=
```
Clear[sortujr];
sortujr[{a___, x_, y_, b___}] := sortujr[{a, y, x, b}] /; x > y;
sortujr[x_] := x;
```

In[270]:=
```
Clear[sortujm];
sortujm[{a___, x_, y_, b___}] := sortujm[{a, y, x, b}] /; x < y;
sortujm[x_] := x;
```

Pierwsza definicja będzie korzystać z instrukcji OptionsPattern i OptionValue:

In[273]:=
```
Clear[sortowanie1];
Options[sortowanie1] = {rosnaco → True};
sortowanie1[a_List, OptionsPattern[]] := Module[{wybor, wynik},
  wybor = OptionValue[rosnaco];
  If[wybor,
       wynik = sortujr[a],
       wynik = sortujm[a],
       Print["Błędna wartość opcji rosnaco"]; Return[]];
  Return[wynik]
 ]
```

A teraz definicja w „starym stylu” z bezpośrednim odczytem wartości:

```
In[276]:= Clear[sortowanie2];
          Options[sortowanie2] = {rosnaco → True};
          sortowanie2[a_List, opcje___] := Module[{wybor, wynik},
            wybor = rosnaco /. {opcje} /. Options[sortowanie2];
            If[wybor,
                 wynik = sortujr[a],
                 wynik = sortujm[a],
                 Print["Błędna wartość opcji rosnaco"]; Return[]];
            Return[wynik]
           ]
```

Utwórzmy listę złożoną z dwudziestu liczb całkowitych.

```
In[279]:= v = Table[RandomInteger[{0, 100}], {i, 1, 20}]
Out[279]= {42, 60, 15, 17, 34, 75, 69, 4, 72, 100, 44, 61, 59, 12, 54, 91, 47, 74, 40, 16}
```

Posortujmy elementy listy rosnąco:

```
In[280]:= sortowanie1[v]
Out[280]= {4, 12, 15, 16, 17, 34, 40, 42, 44, 47, 54, 59, 60, 61, 69, 72, 74, 75, 91, 100}

In[281]:= sortowanie2[v]
Out[281]= {4, 12, 15, 16, 17, 34, 40, 42, 44, 47, 54, 59, 60, 61, 69, 72, 74, 75, 91, 100}

In[282]:= sortowanie1[v, rosnaco → True]
Out[282]= {4, 12, 15, 16, 17, 34, 40, 42, 44, 47, 54, 59, 60, 61, 69, 72, 74, 75, 91, 100}

In[283]:= sortowanie2[v, rosnaco → True]
Out[283]= {4, 12, 15, 16, 17, 34, 40, 42, 44, 47, 54, 59, 60, 61, 69, 72, 74, 75, 91, 100}
```

oraz malejąco:

```
In[284]:= sortowanie1[v, rosnaco → False]
Out[284]= {100, 91, 75, 74, 72, 69, 61, 60, 59, 54, 47, 44, 42, 40, 34, 17, 16, 15, 12, 4}

In[285]:= sortowanie2[v, rosnaco → False]
Out[285]= {100, 91, 75, 74, 72, 69, 61, 60, 59, 54, 47, 44, 42, 40, 34, 17, 16, 15, 12, 4}
```

Przy innej wartości opcji rosnaco otrzymamy komunikat o błędzie.

```
In[286]:= sortowanie1[v, rosnaco → www]
          Błędna wartość opcji rosnaco
In[287]:= sortowanie2[v, rosnaco → www]
          Błędna wartość opcji rosnaco
```

2.8. Atrybuty

```
In[288]:= Clear["Global`*"]
```

Wszystkie funkcje i instrukcje programu *Mathematica* mogą posiadać odpowiednie atrybuty. Standardowo wszystkie nowe funkcje nie są wyposażone w żadne atrybuty. Atrybuty każdego symbolu można obejrzeć, definiować oraz usuwać za pomocą instrukcji:

```
     Attributes[instrukcja]              wyświetla atrybuty podanej instrukcji;
  SetAttributes[funkcja, atrybut]        ustawia nowy atrybut dla podanej funkcji;
ClearAttributes[funkcja, atrybut]          pozbawia funkcję podanego atrybutu;
          ClearAll[f]              odwołanie definicji funkcji f i usunięcie przypisanych
```

```
                                    jej atrybutów, gdy funkcja nie posiada atrybutu Protected.
```

Zobaczmy, jakie atrybuty posiada instrukcja przypisania (Set lub =).

```
In[289]:= Attributes[Set]
Out[289]= {HoldFirst, Protected, SequenceHold}

In[290]:= x = 1; y = 2;

In[291]:= Set[x, y]
Out[291]= 2
```

Atrybut HoldFirst zabrania wyliczania pierwszego argumentu. Zobaczmy, co się stanie, gdy wyliczymy ten argument (służy do tego instrukcja Evaluate)

```
In[292]:= Set[Evaluate[x], y]
          Set: Cannot assign to raw object 2.
Out[292]= 2
```

Domyślnie drugi argument jest wyliczany. Zabrońmy jego wyliczenia.

```
In[293]:= Set[x, HoldForm[y]]
Out[293]= y

In[294]:= {x, y}
Out[294]= {y, 2}
```

Instrukcje wykorzystane w powyższym przykładzie.

```
Evaluate[wyrażenie] nakazuje obliczenie wyrażenia;
HoldForm[wyrażenie] zakazuje obliczenie wyrażenia.
```

Dzięki atrybutowi SequenceHold, możemy wykonywać tego typu podstawienia:

```
In[295]:= Set[{a, b}, {2, 3}]
Out[295]= {2, 3}

In[296]:= a
Out[296]= 2

In[297]:= b
Out[297]= 3
```

Wybrane atrybuty.

```
   Constant                                  funkcja stała;
     Flat                      funkcja łączna (f[x, f[y, z]] = f[f[x, y], z]);
  Orderless                        funkcja przemienna (f[x, y] = f[y, x]);
 OneIdentity                               funkcja tożsamościowa;
   HoldAll                      wszystkie argumenty pozostają niewyliczone;
  HoldFirst                      pierwszy argument pozostaje niewyliczony;
   HoldRest              wszystkie poza pierwszym argumentem pozostają niewyliczone;
SequenceHold             argumenty będące ciągami pozostają w postaci niewyliczonej;
   Listable                        zakres funkcji jest rozszerzony do list;
    Locked                      atrybuty funkcji nie mogą zostać zmienione;
  Protected                         funkcja nie może być modyfikowana;
ReadProtected                     definicja funkcji nie może być wyświetlona;
  Temporary    funkcja (lub symbol) jest lokalana i usuwana, gdy przez dłuższy czas nie jest używana.
```

Zdefiniujmy nową funkcję.

In[298]:=
```
Clear[f];
f[x_, y_] := x + y + 2
```

Sprawdźmy, jakie posiada atrybuty.

In[300]:=
```
Attributes[f]
```

Out[300]= { }

Zapytajmy o definicję symbolu f.

In[301]:=
```
?f
```

Out[301]=

Symbol

Global`f

Definitions

```
f[x_, y_] := x + y + 2
```

Full Name Global`f

Nadajmy funkcji f atrybut zabraniający odczytu jej definicji.

In[302]:=
```
SetAttributes[f, ReadProtected]
```

Zobaczmy jaką teraz otrzymamy odpowiedź na zapytanie o symbol f.

In[303]:=
```
?f
```

Out[303]=

Symbol

Global`f

Attributes {ReadProtected}

Full Name Global`f

W dalszym ciągu możemy zmieniać definicję funkcji f.

In[304]:=
```
f[0, 0] := 0
```

In[305]:=
```
{f[0, 0], f[1, 2]}
```

Out[305]= {0, 5}

Zabezpieczmy funkcję przed jakimikolwiek zmianami.

In[306]:=
```
SetAttributes[f, Protected]
```

Spróbujmy wprowadzić zmiany w definicji funkcji.

In[307]:=
```
f[1, 1] := -10
```

SetDelayed: Tag f in f[1, 1] is Protected.

Out[307]= $Failed

Sprawdźmy atrybuty funkcji.

In[308]:=
```
Attributes[f]
```

Out[308]= {Protected, ReadProtected}

Usuńmy nadane atrybuty.

In[309]:=
```
ClearAll[f]
```

ClearAll: Symbol f is Protected.

In[310]:=
```
?f
```

Out[310]=

Symbol

Global`f

Attributes {Protected, ReadProtected}
Full Name Global`f

In[311]:=
```
f[3, 4]
```

Out[311]= 9

In[312]:=
```
ClearAttributes[f, Protected];
```

In[313]:=
```
Attributes[f]
```

Out[313]= {ReadProtected}

In[314]:=
```
ClearAll[f];
```

In[315]:=
```
Attributes[f]
```

Out[315]= { }

In[316]:=
```
?f
```

Out[316]=

Symbol

Global`f

Full Name Global`f

In[317]:=
```
f[3, 2]
```

Out[317]= f[3, 2]

2.9. Funkcje prymitywne

In[318]:=
```
Clear["Global`*"]
```

Do tej pory mieliśmy kontakt wyłącznie z tradycyjnymi funkcjami programu *Mathematica*. Są one bardzo podobne do ich odpowiedników w takich językach programowania jak C, czy Pascal. Poza nimi istnieje jeszcze jeden sposób definiowania funkcji - są to tak zwane funkcje prymitywne (pure function). Bywają one także nazywane funkcjami anonimowymi, ponieważ nie mają nazw własnych. Funkcje prymitywne można zapisać wykorzystując instrukcje:

```
Function[argument, wzór]                    definicja funkcji prymitywnej;
         wzór &                             definicja funkcji prymitywnej;
   #(lub #1, #2, ...)    odwołanie do argumentu dla funkcji jednejzmiennej (lub wieluzmiennych).
```

Zdefiniujmy funkcję prymitywną i policzmy jej wartość w wybranym punkcie.

In[319]:=
```
Function[t, t^2 + 1][3]
```

Out[319]= 10

To samo w inny sposób.

In[320]:=
```
(#^2 + 1) &[3]
```

Out[320]= 10

Można również definiować funkcje wielu zmiennych. Dla takich funkcji jej kolejne argumenty mają oznaczenia #NumerArgumentu. Tak więc #1 określa pierwszy argument, #2 - drugi itd.

```
In[321]:= (#1 - #2)^3 &[1, 3]

Out[321]= -8
```

Zaletą funkcji prymitywnych, poza ekonomicznym zapisem, jest także fakt, że nie musimy wykorzystywać dodatkowych symboli dla nazw funkcji. Funkcje prymitywne przecież nie mają nazw.

Funkcje prymitywne mogą być wykorzystane na przykład do określania warunków, jakie muszą spełniać argumenty funkcji.

Zdefiniujmy funkcję, która będzie działała tylko dla argumentów podzielnych przez pięć.

```
In[322]:= Clear[f];
          f[x_?((Mod[#1, 5] == 0) &)] := x

In[324]:= f[2]

Out[324]= f[2]

In[325]:= f[10]

Out[325]= 10
```

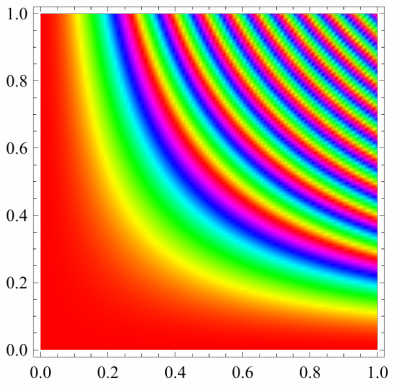
Wykorzystanie drugiej postaci funkcji prymitywnej.

```
In[326]:= Clear[g];
          g[x_?(Function[t, Mod[t, 5] == 0])] := x

In[328]:= {g[14], g[155]}

Out[328]= {g[14], 155}
```

Innym przykładem wykorzystania funkcji prymitywnych jest zastosowanie ich jako wartość opcji. Na przykład przy dobieraniu koloru na rysunku.

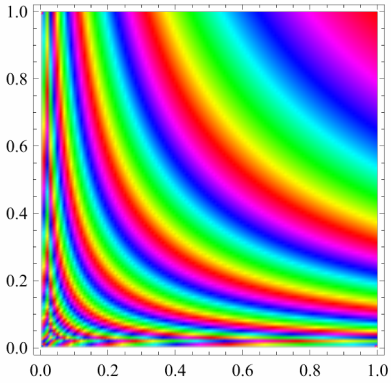
```
In[329]:= DensityPlot[Sin[x * y], {x, 0, 1}, {y, 0, 1}, Mesh → False, PlotPoints → 100,
           ColorFunction → ((Hue[10 * #1^2]) &), ImageSize → 200]
```

Out[329]=



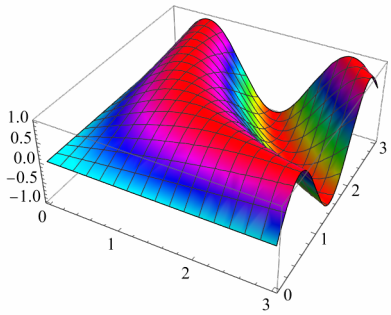
In[330]:=
```
DensityPlot[Sin[x * y], {x, 0, 1}, {y, 0, 1}, Mesh → False, PlotPoints → 100,
 ColorFunction → ((Hue[Log[Abs[#1]]]) &), ImageSize → 200]
```

Out[330]=



In[331]:=
```
Plot3D[Sin[x * y], {x, 0, 3}, {y, 0, 3},
 ColorFunction → Function[{x, y, z}, Hue[z]], ImageSize → 200]
```

Out[331]=



2.10. Kompilacja funkcji

In[332]:=
```
Clear["Global`*"]
```

Szczególnie niekorzystny wpływ na szybkość działania programów napisanych w języku *Wolfram* ma fakt, że każda z funkcji, każde z wyrażeń jest interpretowane w momencie wykonania. Powoduje to znaczne wydłużenie czasu działania programów o dużej liczbie obliczeń. Duże przyspieszenie można uzyskać dzięki używaniu wcześniej skompilowanych funkcji. W programie *Mathematica* tego typu funkcje można otrzymać, wykorzystując instrukcję Compile.

```
    Compile[{arg1, ...}, wyrażenie]     tworzy skompilowaną funkcję, która oblicza podane wyrażenie,
                                            wykorzystując wartości numeryczne każdego argumentu;
Compile[{{arg1, typ1}, ...}, wyrażenie]      jw., zakładając, że argument arg1 jest typu typ1 itd.
```

Wykorzystać możemy następujące typu argumentów:

```
  _Integer    całkowity;
   _Real     rzeczywisty;
 _Complex    zespolony;
True | False  logiczny.
```

Zobaczmy przykład

In[333]:=
```
Timing[t = 0.81; Do[t = Sin[t] * (1 - t^2), {i, 1, 1000000}]; t]
```

Out[333]=
```
{1.45313, 0.000654649}
```

In[334]:=
```
cf = Compile[{x, {n, _Integer}}, Module[{t, i}, t = x;
     Do[t = Sin[t] * (1 - t^2), {i, 1, n}];
     Return[t]]];
```

```
In[335]:= Timing[cf[0.81, 1000000]]
Out[335]= {0.09375, 0.000654649}
```

W powyższym przypadku funkcja skompilowana wykonała obliczenia o wiele szybciej. W przypadku jeszcze większej liczby obliczeń dysproporcja ta byłaby znacznie większa. Dlatego dla takich przypadków zalecane jest używanie wyłącznie skompilowanych funkcji. Czasami zdarza się jednak, że niestety nie możemy skorzystać z tego udogodnienia. Dzieje się tak np. wtedy, gdy chcemy modyfikować dokładność poszczególnych obliczeń. Wiąże się to z tym, że raz skompilowanej funkcji nie można później w żaden sposób modyfikować. Dodatkową wadą jest fakt, że funkcje wykorzystujące liczby wymagające znacznej rezerwacji pamięci (np. precyzyjne lub bardzo duże) nie są wykonywane w postaci skompilowanej - dla tych funkcji, po przekroczeniu zakresu, *Mathematica* automatycznie przechodzi na postać nieskompilowaną.

Jeśli kompilowana instrukcja ma atrybut Listable, to *Mathematica* automatycznie przygotuje ją do obliczeń równoległych.

```
In[336]:= cP1 = Compile[{{x}},
              Module[{sum = 1.0, inc = 1.0}, Do[inc = inc * x / i;
                sum = sum + inc, {i, 10000}]; sum],
              RuntimeAttributes → {Listable}];
            arg = Range[ -50., 50, 0.02];
            cP1[arg]; // AbsoluteTiming
Out[338]= {1.15158, Null}

In[339]:= cP = Compile[{{x}},
              Module[{sum = 1.0, inc = 1.0}, Do[inc = inc * x / i;
                sum = sum + inc, {i, 10000}]; sum],
              RuntimeAttributes → {Listable}, Parallelization → True];
            cP[arg]; // AbsoluteTiming
Out[340]= {1.22852, Null}

In[341]:= cS = Compile[{{x}},
              Module[{sum = 1.0, inc = 1.0}, Do[inc = inc * x / i;
                sum = sum + inc, {i, 10000}]; sum],
              RuntimeAttributes → {Listable}, Parallelization → False];
            cS[arg]; // AbsoluteTiming
Out[342]= {2.84648, Null}

In[343]:= $ProcessorCount
Out[343]= 4
```

2.11. Dodatkowe instrukcje

```
In[344]:= Clear["Global`*"]
```

Mathematica pozwala nam traktować duże struktury (wektory, macierze, listy) tak jak proste obiekty. Unikamy w ten sposób zbędnych pętli powszechnych w tradycyjnych językach programowania (np. Fortran). W taki sposób działają wszystkie instrukcje posiadające atrybut Listable, np. instrukcja Sin:

```
In[345]:= Sin[{π, 2.1, π/4, 45 °}]
Out[345]= {0, 0.863209, 1/√2, 1/√2}

In[346]:= Attributes[Sin]
Out[346]= {Listable, NumericFunction, Protected}
```

W takich przypadkach odpowiednia instrukcja jest wykonywana dla każdego elementu listy. Jeśli mamy funkcję nie posiadającą atrybutu Listable i chcemy, aby została wykonana dla każdego elementu, to nie musimy zaczynać od pisania pętli lub niepotrzebnie ingerować

w funkcję, zmieniając jej atrybuty. Istnieje instrukcja, która wymusza działanie wszystkich funkcji (bez względu na ich atrybuty) na listach. -

```
    Map[funkcja, lista]              wymusza działanie funkcji na elementach listy;
      funkcja /@ lista                                        jw.;
Map[funkcja, lista, poziom] wymusza działanie funkcji na elementach listy z zadanego poziomu;
  MapAt[funkcja, lista, n]          wymusza działanie funkcji na n-tym elemencie listy.
```

Przykładem instrukcji nieposiadającej atrybutu Listable jest instrukcja IntegerQ sprawdzająca, czy dane wyrażenie jest liczbą całkowitą. Zobaczmy, jakie są atrybuty tej instrukcji:

```
In[347]:= Attributes[IntegerQ]
Out[347]= {Protected}
```

Próba użycia instrukcji dla listy skończy się informacją, że podany element nie jest liczbą całkowitą, bez wnikania w strukturę wewnętrzną listy.

```
In[348]:= IntegerQ[{2, 4.5, 2 + 3 i, 8}]
Out[348]= False
```

Natomiast gdy wykorzystamy instrukcję Map, będziemy mogli zagłębić się do wnętrza podanej listy.

```
In[349]:= Map[IntegerQ, {2, 4.5, 2 + 3 i, 8}]
Out[349]= {True, False, False, True}

In[350]:= IntegerQ /@ {2, 4.5, 2 + 3 i, 8}
Out[350]= {True, False, False, True}
```

W powyższym przykładzie jako argumentu użyliśmy jednowymiarowej struktury, jaką jest lista. Co się stanie, gdy użyjemy argumentu będącego macierzą? Zobaczmy:

```
In[351]:= Map[IntegerQ, {{2, 4.5}, {2 + 3 i, 8}}]
Out[351]= {False, False}
```

Przy powyższym wywołaniu instrukcja IntegerQ została zastosowana do wektorów tworzących wiersze macierzy. Można jednak używać instrukcji Map dla wielowymiarowych struktur. Należy tylko wtedy podać jako ostatni argument numer poziomu struktury, do którego chcemy zastosować instrukcję. Standardowo tym poziomem jest poziom pierwszy.

```
In[352]:= Map[IntegerQ, {{2, 4.5}, {2 + 3 i, 8}}, {2}]
Out[352]= {{True, False}, {False, True}}
```

Zdefiniujmy funkcję i zastosujmy ją do wybranego elementu listy.

```
In[353]:= f[x_] := x^2
In[354]:= MapAt[f, {1, -2, 3, 4}, 2]
Out[354]= {1, 4, 3, 4}
```

Instrukcje Map i Apply (która zostanie omówiona poniżej) są dość często wykorzystywane w połączeniu z funkcjami prymitywnymi. Ułatwiają one zapis bardzo skomplikowanych funkcji, jak również działają szybciej od tradycyjnych odpowiedników.
Napiszmy procedurę, która każdy element ujemny wektora podnosi do kwadratu, natomiast pozostałe pozostawia bez zmian.

```
In[355]:= Map[If[#1 < 0, #1^2, #1] &, {1, 2, -3, 4, -4, -7}]
Out[355]= {1, 2, 9, 4, 16, 49}
```

Jeszcze jeden przykład.

```
In[356]:= Map[If[#1 > 0, 1. * Log[#1], 0] &, {1, 2, -3, 4, -4, 0}]
Out[356]= {0., 0.693147, 0, 1.38629, 0, 0}
```

To samo, ale trochę inaczej.

```
In[357]:= If[#1 > 0, Log[N@#1], 0] & /@ {1, 2, -3, 4, -4, 0}
Out[357]= {0., 0.693147, 0, 1.38629, 0, 0}
```

Zapis f@x odpowiada zapisowi f[x].

Inną równie użyteczną instrukcją przy przetwarzaniu struktur jest instrukcja Apply. Jej użyteczność wynika z tego, że w zasadzie każdy element języka (struktury danych, pętle, instrukcje warunkowe, funkcje) jest traktowany przez system jednakowo, tzn. wszystko jest przechowywane w pamięci w jednolitej formie zgodnej z formatem funkcji. I tak np. wyrażenie a+b jest przechowywane jako Plus[a,b], natomiast a+b*c jako Plus[a,Times[b,c]].

```
Apply[nagłówek, wyrażenie]                zmienianagłówekwyrażenia;
nagłówek @@ wyrażenie                     jw.;
Apply[nagłówek, wyrażenie, poziomy]       zmienianagłówekdlapodanychpoziomówwyrażenia;
nagłówek @@@ wyrażenie                    to samo co Apply[nagłówek, wyrażenie, {1}], czyli
                                          zmienia nagłówek dla piewszego poziomu wyrażenia;
MapApply[nagłówek, wyrażenie]             jw.
```

O postaci danego wyrażenia można się przekonać dzięki instrukcji FullForm.

```
In[358]:= FullForm[a + b * c]
Out[358]//FullForm= Plus[a, Times[b, c]]
```

Jak widać, wyrażenie wynikowe jest podobne do zwykłej funkcji - ma swój nagłówek będący nazwą i argumenty. Identycznie wyglądają nawet najbardziej skomplikowane struktury danych.
O znaczeniu instrukcji Apple świadczy następujący przykład. Spróbujmy znaleźć iloczyn wszystkich elementów pewnej listy. Jak się do tego zabrać? Od razu narzuca się napisanie pętli mnożącej poszczególne elementy tablicy. Powiedzmy, że mamy następującą listę elementów:

```
In[359]:= dane = {2, a, 11, b, 3, d}
Out[359]= {2, a, 11, b, 3, d}
```

Możemy policzyć iloczyn, korzystając z instrukcji Product.

```
In[360]:= w = ∏_{i=1}^{Length[dane]} dane[[i]]
Out[360]= 66 a b d
```

Porównajmy zapis pełnej formy ostatniego wyrażenia z zapisem listy.

```
In[361]:= FullForm[w]
Out[361]//FullForm= Times[66, a, b, d]
In[362]:= FullForm[dane]
Out[362]//FullForm= List[2, a, 11, b, 3, d]
```

Widać, że wystarczy zamienić nagłówki (czyli użyć instrukcji Apply) i otrzymamy właściwy wynik. Oto najkrótszy sposób zrealizowania mnożenia elementów listy:

```
In[363]:= Apply[Times, dane]
Out[363]= 66 a b d
In[364]:= Times @@ dane
Out[364]= 66 a b d
```

Podobnie można zrealizować dodawanie elementów listy:

```
In[365]:= Apply[Plus, dane]
Out[365]= 16 + a + b + d
```

```
In[366]:= Plus @@ dane
Out[366]= 16 + a + b + d
```

ale do sumowania elementów listy możemy wykorzystać także instrukcję Total.

```
In[367]:= Total[dane]
Out[367]= 16 + a + b + d
```

Zdefiniujmy macierz.

```
In[368]:= a = Table[RandomReal[], {i, 1, 10}, {j, 1, 5}];
```

Zastosowanie instrukcji Apply z dwoma argumentami i nagłówkiem Plus, spowoduje wysumowanie elementów każdej kolumny podanej macierzy.

```
In[369]:= w = Apply[Plus, a]
Out[369]= {4.71356, 5.46174, 5.08664, 5.86418, 5.70533}
```

Jeśli chcemy wysumować wszystkie elementy macierzy, możemy postąpić następująco:

```
In[370]:= Apply[Plus, a, {0, 1}]
Out[370]= 26.8315
```

Suma elementów w poszczególnych wierszach:

```
In[371]:= Apply[Plus, a, 2]
Out[371]= {2.86183, 2.53804, 2.41257, 2.66835, 2.08764, 3.38652, 2.99696, 1.35666, 3.49838, 3.02452}
```

Instrukcja Total także wysumuje kolumny macierzy:

```
In[372]:= Total[a]
Out[372]= {4.71356, 5.46174, 5.08664, 5.86418, 5.70533}
```

Suma wszystkich elementów macierzy:

```
In[373]:= Total[a, 2]
Out[373]= 26.8315
```

Wysumowanie wierszy macierzy:

```
In[374]:= Total[a, {2}]
Out[374]= {2.86183, 2.53804, 2.41257, 2.66835, 2.08764, 3.38652, 2.99696, 1.35666, 3.49838, 3.02452}
```

lub korzystając z transpozycji macierzy:

```
In[375]:= Total[Transpose[a]]
Out[375]= {2.86183, 2.53804, 2.41257, 2.66835, 2.08764, 3.38652, 2.99696, 1.35666, 3.49838, 3.02452}
```

Różnice użycia @, @@, @@@ oraz /@.

```
In[376]:= h@x
Out[376]= h[x]
In[377]:= h@{x, y, z}
Out[377]= h[{x, y, z}]
In[378]:= h /@ {x, y, z}
Out[378]= {h[x], h[y], h[z]}
```

```
In[379]:= h @@ {x, y, z}
Out[379]= h[x, y, z]
In[380]:= h @@ {{x, y, z}, {v1, v2, v3}}
Out[380]= h[{x, y, z}, {v1, v2, v3}]
In[381]:= h @@@ {{x, y, z}, {v1, v2, v3}}
Out[381]= {h[x, y, z], h[v1, v2, v3]}
In[382]:= h @@@ {x, y, z}
Out[382]= {x, y, z}
In[383]:= h @@@ {{x, y, z}}
Out[383]= {h[x, y, z]}
```

Kolejne instrukcje ułatwiają tworzenie iteracji.

```
Nest[funkcja, wyrażenie, n]          n razy stosuje funkcję do wyrażenia;
NestList[funkcja, wyrażenie, n]      tworzy listę, której kolejnymi elementami są
                                     wyniki zastosowania podanej funkcji do wyrażenia;
NestWhile[funkcja, wyrażenie, test]  stosuje funkcję do wyrażenia tak długo, jak test daje wartość True.
```

Zobaczmy przykłady.

```
In[384]:= Nest[Sin, x, 3]
Out[384]= Sin[Sin[Sin[x]]]
In[385]:= NestList[Sin, x, 3]
Out[385]= {x, Sin[x], Sin[Sin[x]], Sin[Sin[Sin[x]]]}
```

Możemy także wykorzystywać funkcje prymitywne.

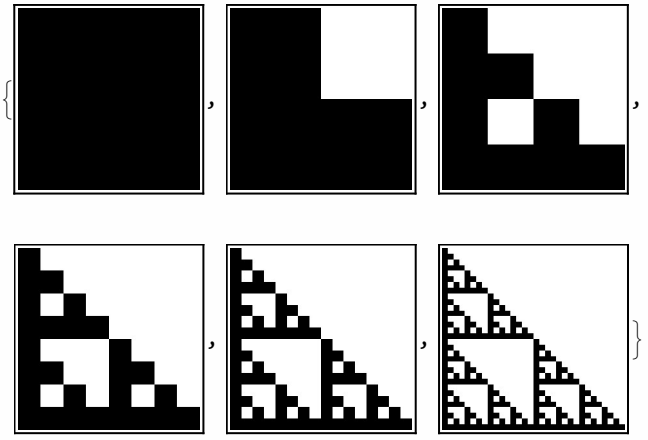
```
In[386]:= c = 0.2 + 0.12 i;
          Nest[(#^2 + c) &, 0, 30]
Out[387]= 0.2 + 0.2 i
In[388]:= NestWhile[Log, 100, # > 0 &]
Out[388]= Log[Log[Log[Log[100]]]]
In[389]:= NestWhile[# / 2 &, 123456, EvenQ]
Out[389]= 1929
```

Ciąg macierzy, których obraz graficzny przybliża trójkąt Sierpińskiego.

```
In[390]:= NestList[ArrayFlatten[{{#, 0}, {#, #}}] &, {{1}}, 2]
Out[390]= {{{1}}, {{1, 0}, {1, 1}}, {{1, 0, 0, 0}, {1, 1, 0, 0}, {1, 0, 1, 0}, {1, 1, 1, 1}}}
```

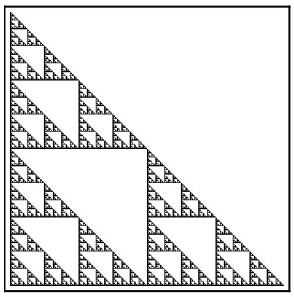
```
In[391]:= SetOptions[ArrayPlot, ImageSize → 100];
          ArrayPlot /@ NestList[ArrayFlatten[{{#, 0}, {#, #}}] &, {{1}}, 5]
```

Out[392]=



```
In[393]:= SetOptions[ArrayPlot, ImageSize → 150];
          ArrayPlot[Nest[ArrayFlatten[{{#, 0}, {#, #}}] &, {{1}}, 8]]
```

Out[393]=

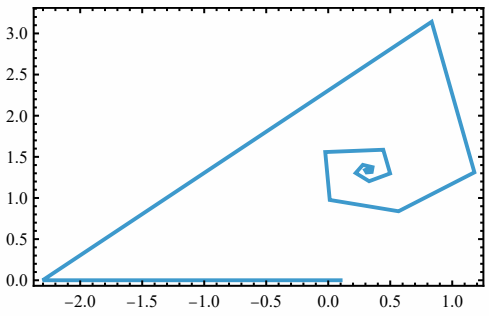


```
FixedPoint[funkcja, wyrażenie]     stusuje funkcję do wyrażenia tak długo, jak wynik ulega zmianie;
FixedPoint[funkcja, wyrażenie, n]                jw., wykonuje co najwyżej n iteracji;
FixedPointList[funkcja, wyrażenie]                          tworzy listę.
```

Zobaczmy przykład.

```
In[394]:= FixedPoint[Log, 0.1]
Out[394]= 0.318132 + 1.33724 i
In[395]:= Log[%]
Out[395]= 0.318132 + 1.33724 i
In[396]:= ListPlot[({Re[#1], Im[#1]} &) /@ FixedPointList[Log, 0.1],
           Joined → True, PlotRange → All, Frame → True, Axes → False, ImageSize → 250]
```

Out[396]=



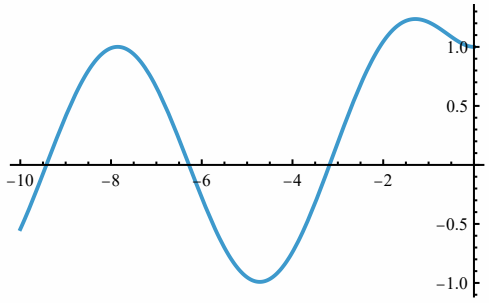
Możemy wykorzystać instrukcję FixedPoint do napisania procedury znajdującej pierwiastek funkcji metodą Newtona.

In[397]:= `metnewtona[f_, pocz_] := FixedPoint[(# - f[#]/f'[#]) &, pocz];`

In[398]:= `g[x_] := Exp[x] - Sin[x]`

In[399]:= `Plot[g[x], {x, -10, 0}, ImageSize → 250]`

Out[399]=



In[400]:= `x0 = metnewtona[g, -4.]`

Out[400]= `-3.18306`

In[401]:= `g[x0]`

Out[401]= 2.15106×10^{-16}

In[402]:= `x01 = metnewtona[g, -4.`200]`

Out[402]= `-3.183063011933363591939186995636394557681148812158796678009742072309356405022987908245518550981967505446887446472393976453831070251333726151136744667351172338302147865878235660558829017529641299230`5

In[403]:= `g[x01]`

Out[403]= $0. \times 10^{-197}$

In[404]:= `x02 = metnewtona[g, -6.`50]`

Out[404]= `-6.2813143662107954869842210466630192509299809815`4

In[405]:= `g[x02]`

Out[405]= $0. \times 10^{-48}$

2.12. Tworzenie środowisk

In[406]:= `Clear["Global`*"]`

Bardzo często wygodnie jest podzielić projekt, nad którym pracujemy, na mniejsze części. *Mathematica* udostępnia nam w tym celu mechanizm kontekstów. Kontekst jest środowiskiem, w którym przeprowadzamy obliczenia.

```
    $Context          zmienna systemowa, przechowująca nazwę aktualnego kontekstu;
  $ContextPath    zmienna systemowa, zawierająca nazwy aktualnie widzianych kontekstów;
Begin[''nazwa`'']                    tworzy nowy kontekst chwilowy;
      End[]            zamyka kontekst chwilowy, odtwarza poprzednie środowisko,
                           usuwa zdefiniowane w kontekscie zmienne i definicje.
```

Zobaczmy, jakie mamy aktualnie zdefiniowane środowiska.

In[407]:= `$ContextPath`

Out[407]= `{ColorDataVisColors`, System`, Global`}`

Wypiszmy nazwę bieżącego kontekstu, a następnie zdefiniujmy własny o nazwie pomocniczy.

```
In[408]:= $Context
Out[408]= Global`
In[409]:= $Context = "pomocniczy`"
Out[409]= pomocniczy`
```

Zdefiniujmy teraz funkcję i policzmy jej wartość w wybranym punkcie.

```
In[410]:= Clear[ffk];
          ffk[t_] := t^2 + 1;
In[412]:= ffk[2]
Out[412]= 5
```

Powrócimy teraz do środowiska Global`. Można w nim korzystać ze zdefiniowanej funkcji ffk, lecz wymaga to odpowiedniego odwołania się do niej.

```
In[413]:= $Context = "Global`"
Out[413]= Global`
In[414]:= ?? ffk
Out[414]= Missing[UnknownSymbol, ffk]
In[415]:= pomocniczy`ffk[3]
Out[415]= 10
In[416]:= ?? pomocniczy`ffk
```

Out[416]=

Symbol

pomocniczy`ffk

Definitions

pomocniczy`ffk[t_] := $t^2 + 1$

Full Name pomocniczy`ffk

Możemy na stałe udostępnić nowy kontekst, wpisując jego nazwę do zmiennej $ContextPath.

```
In[417]:= AppendTo[$ContextPath, "pomocniczy`"]
Out[417]= {ColorDataVisColors`, System`, Global`, pomocniczy`}
In[418]:= ?? ffk
```

Out[418]=

Symbol

pomocniczy`ffk

Definitions

ffk[t_] := $t^2 + 1$

Full Name pomocniczy`ffk

```
In[419]:= ffk[7]
Out[419]= 50
```

3. Instrukcje wejścia-wyjścia

3.1. Katalogi

In[420]:=
```
Clear["Global`*"]
```

Język *Wolfram* posiada równie rozbudowany zestaw instrukcji zarządzających plikami, jak i wiele innych bardziej ukierunkowanych na zarządzanie zbiorami języków programistycznych. *Mathematica,* podobnie jak każdy system operacyjny, posiada własną ścieżkę wyszukiwania zbiorów. Można ją dość łatwo obejrzeć, gdyż jest ona zapisana w zmiennej $Path.

```
$Path                                   ścieżka wyszukiwania zbiorów;
$Path = Append[$Path, "''katalog''"]    dodanie katalogu do ścieżki wyszukiwania zbiorów;
SetDirectory["''katalog''"]             ustawienie katalogu roboczego;
SetDirectory[NotebookDirectory[]]       ustawienie jako katalog roboczy katalogu,
                                        w którym znajduje się bieżący notatnik;
Directory[]                             podaje nazwę aktualnego katalogu roboczego;
ResetDirectory[]                        powoduje powrót do poprzedniego katalogu roboczego;
ParentDirectory[]                       podaje katalog nadrzędny w stosunku do aktualnego katalogu roboczego.
```

Zobaczmy, jakie katalogi znajdują się w ścieżce wyszukiwania zbiorów.

In[421]:=
```
$Path
```

Out[421]=
```
{C:\Users\damian.slota\AppData\Roaming\Wolfram\DocumentationIndices,
 D:\Wolfram Research\Wolfram\14.2\SystemFiles\Links,
 C:\Users\damian.slota\AppData\Roaming\Wolfram\Kernel,
 C:\Users\damian.slota\AppData\Roaming\Wolfram\Autoload,
 C:\Users\damian.slota\AppData\Roaming\Wolfram\Applications, C:\ProgramData\Wolfram\Kernel,
 C:\ProgramData\Wolfram\Autoload, C:\ProgramData\Wolfram\Applications, .,
 C:\Users\damian.slota, D:\Wolfram Research\Wolfram\14.2\AddOns\Packages,
 D:\Wolfram Research\Wolfram\14.2\SystemFiles\Autoload,
 D:\Wolfram Research\Wolfram\14.2\AddOns\Autoload,
 D:\Wolfram Research\Wolfram\14.2\AddOns\Applications,
 D:\Wolfram Research\Wolfram\14.2\AddOns\ExtraPackages,
 D:\Wolfram Research\Wolfram\14.2\SystemFiles\Kernel\Packages,
 D:\Wolfram Research\Wolfram\14.2\Documentation\English\System,
 D:\Wolfram Research\Wolfram\14.2\SystemFiles\Data\ICC,
 D:\Wolfram Research\Wolfram\14.2\Documentation\ChineseSimplified\System\}
```

Sprawdźmy, jaki katalog jest aktualnie katalogiem roboczym.

In[422]:=
```
Directory[]
```

Out[422]=
```
C:\Users\damian.slota
```

Zmieńmy go.

In[423]:=
```
SetDirectory["c:\\temp"]
```

Out[423]=
```
c:\temp
```

In[424]:=
```
SetDirectory[NotebookDirectory[]]
```

Out[424]=
```
D:\Damian\zajecia\szkolenie-f2s\minut-2
```

Jego katalogiem nadrzędnym jest:

In[425]:=
```
ParentDirectory[]
```

Out[425]=
```
D:\Damian\zajecia\szkolenie-f2s
```

Powróćmy do poprzedniego katalogu roboczego.

```
In[426]:= ResetDirectory[]
Out[426]= c:\temp
In[427]:= Directory[]
Out[427]= c:\temp
```

3.2. Pliki

```
In[428]:= Remove["Global`*"]
```

Instrukcja FileNames pozwala wypisać nazwy plików znajdujących się w wybranym katalogu.

FileNames[]	wypisuje nazwy plików i katalogów znajdujących się w aktualnym katalogu roboczym;
FileNames[''forma'']	wypisuje nazwy plików znajdujących się w aktualnym katalogu roboczym, pasujące do zadanej formy;
FileNames[''forma'', ''katalog'']	wypisuje nazwy plików znajdujących się w zadanym katalogu, pasujące do zadanej formy;
FileNames[''forma'', ''katalog'', n]	jw., przeszukiwane są także podkatalogi do n - tego poziomu;
FileNames[''forma'', ''katalog'', Infinity]	jw., przeszukiwane są wszystkie podkatalogi.

Zobaczmy, jak działa opisana instrukcja.

```
In[429]:= SetDirectory[NotebookDirectory[]]
Out[429]= D:\Damian\zajecia\szkolenie-f2s\minut-2
In[430]:= FileNames[]
Out[430]= {20240717_135007.jpg, bbb.pdf, Elementy-jezyka-Wolfram-minut-bez-tytulu.nb,
  Elementy-jezyka-Wolfram-minut-full.nb, Elementy-jezyka-Wolfram-minut.nb,
  Elementy-jezyka-Wolfram poprawki redakcji.pdf, inne-pliki, rrzm_przekroj_lac.txt, str-tyt}
In[431]:= FileNames["*.jpg"]
Out[431]= {20240717_135007.jpg}
In[432]:= FileNames["*.nb", "d:\\"]
Out[432]= {}
```

Komunikacja z dyskiem jest wykonywana poprzez strumienie. Możemy wykonywać te operacje „krok po kroku", czyli najpierw otworzyć plik (strumień), następnie na nim pracować, a na końcu go zamknąć. W większości zastosowań wystarcza jednak korzystanie z instrukcji, które za nas otwierają i zamykają zbiory. Nie musimy się wtedy zajmować warstwą wewnętrzną komunikacji z dyskiem. Poza tym wbrew pozorom możliwości dostarczane przez te instrukcje nie są ubogie i przy dość dobrej znajomości tych instrukcji mogą się one okazać wystarczające dla większości zastosowań. Oto krótki opis najważniejszych spośród tych instrukcji:

Put[wyrażenie$_1$, ..., ''plik''] lub wyrażenie >> plik	zapisuje wyrażenia do pliku;
PutAppend[wyrażenie$_1$, ..., ''plik''] lub wyrażenie >>> plik	zapisuje wyrażenie, dopisując je na koniec pliku;
Get[''plik''] lub << plik	odczytuje zawartość pliku, wykonując wszystkie zawarte w nim wyrażenia i zwracając wynik ostatniego z nich;
Save[''plik'', symbol$_1$, ...]	zapisuje do pliku definicje podanych symboli;
FindList[''plik'', ''tekst'']	zwraca listę linii w pliku zawierających zadany tekst;
FindList[''plik'', ''tekst'', n]	jw., ale wyświetlanych jest tylko n pierwszych linii;
ReadList[''plik'']	wczytuje wszystkie wyrażenia z pliku i zwraca ich listę;
ReadList[''plik'', typ]	odczytuje wszystkie obiekty zadanego typu i zwraca ich listę;
ReadList[''plik'', {typ$_1$, typ$_2$, ...}]	odczytuje obiekty zgodne z podaną sekwencją;

```
ReadList['' plik '', {typ1, typ2, ...}, n]          wczytuje tylko n pierwszych obiektów.
```

Ustawmy katalog roboczy.

```
In[433]:= SetDirectory[NotebookDirectory[]]
Out[433]= D:\Damian\zajecia\szkolenie-f2s\minut-2
```

Utwórzmy tablicę liczb.

```
In[434]:= wyniki = Table[i^2, {i, 1, 20}]
Out[434]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
```

Zapiszmy ją na dysku.

```
In[435]:= Put[wyniki, "test1.txt"]
```

Wczytajmy zawartość pliku.

```
In[436]:= << test1.txt
Out[436]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
```

Wczytywaną listę możemy zapisać pod zmienną.

```
In[437]:= aa = << test1.txt
Out[437]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}

In[438]:= Total[aa]
Out[438]= 2870
```

Możemy też wypisać zawartość pliku.

```
In[439]:= FilePrint["test1.txt"]

{1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289,
 324, 361, 400}
```

Zapiszemy teraz te same dane bez przecinków i nawiasów.

```
In[440]:= Do[PutAppend[wyniki[[i]], "test2.txt"], {i, 1, Length[wyniki]}]

In[441]:= FilePrint["test2.txt"]

1
4
9
16
25
36
49
64
81
100
121
144
169
196
225
256
289
324
361
400
```

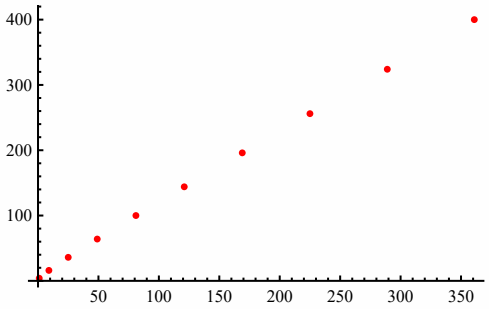
Jeśli chcemy dane zapisane w pliku wykorzystać w dalszych obliczeniach, to możemy do ich odczytu wykorzystać instrukcję ReadList.

```
In[442]:= dane1 = ReadList["test2.txt", Number]
Out[442]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400}
In[443]:= Total[dane1]
Out[443]= 2870
```

Możemy wczytywane dane grupować po dwie, trzy itd.

```
In[444]:= dane2 = ReadList["test2.txt", {Number, Number}]
Out[444]= {{1, 4}, {9, 16}, {25, 36}, {49, 64}, {81, 100},
  {121, 144}, {169, 196}, {225, 256}, {289, 324}, {361, 400}}
In[445]:= ListPlot[dane2, PlotStyle → {PointSize[0.015], RGBColor[1, 0, 0]}, ImageSize → 250]
```

Out[445]=



Zadanie. Napisać procedurę, której argumentami będą nazwa pliku oraz wektor wyników, a która każdy element wektora wyników zapisze w osobnej linii podanego pliku.

Rozwiązanie.

```
In[446]:= Clear[zapisWynikow1];
          zapisWynikow1[plik_, dane_] := Module[{strum, n, i},
             strum = OpenWrite[plik];
             n = Length[dane];
             Do[WriteString[strum, dane[[i]]];
              WriteString[strum, "\n"], {i, 1, n}];

          Close[strum]];
In[448]:= SetDirectory[NotebookDirectory[]]
Out[448]= D:\Damian\zajecia\szkolenie-f2s\minut-2
In[449]:= wyniki1 = Table[i^3, {i, 1, 10}];
In[450]:= zapisWynikow1["wyn1.txt", wyniki1]
Out[450]= wyn1.txt
```

```
In[451]:= FilePrint["wyn1.txt"]
```

```
1
8
27
64
125
216
343
512
729
1000
```

Zadanie. Napisać procedurę, której argumentami będą nazwa pliku oraz macierz wyników (dwie kolumny i dowolna liczba wierszy), a która w jednej linii zapisze elementy jednego wiersza macierzy oddzielone tabulacją.

Rozwiązanie.

```
In[452]:= Clear[zapisWynikow2];
          zapisWynikow2[plik_, dane_] := Module[{strum, n, i},
             strum = OpenWrite[plik];
             n = Length[dane];
             Do[WriteString[strum, dane[[i, 1]]];
              WriteString[strum, "\t"];
              WriteString[strum, dane[[i, 2]]];
              WriteString[strum, "\n"], {i, 1, n}];

          Close[strum]];
```

```
In[454]:= wyniki2 = Table[{i, Sin[i]}, {i, 0, 3, 0.5}]
```

```
Out[454]= {{0., 0.}, {0.5, 0.479426}, {1., 0.841471},
           {1.5, 0.997495}, {2., 0.909297}, {2.5, 0.598472}, {3., 0.14112}}
```

```
In[455]:= zapisWynikow2["wyn2.txt", wyniki2]
```

```
Out[455]= wyn2.txt
```

```
In[456]:= FilePrint["wyn2.txt"]
```

```
0.  0.
0.5 0.479426
1.  0.841471
1.5 0.997495
2.  0.909297
2.5 0.598472
3.  0.14112
```

Przykład odczytu danych z pliku

Ustawmy katalog roboczy.

```
In[457]:= SetDirectory[NotebookDirectory[]]
```

```
Out[457]= D:\Damian\zajecia\szkolenie-f2s\minut-2
```

Zobaczmy, jakie pliki są w katalogu roboczym.

In[458]:=
```
FileNames[]
```

Out[458]=
```
{20240717_135007.jpg, bbb.pdf, Elementy-jezyka-Wolfram-minut-bez-tytulu.nb,
 Elementy-jezyka-Wolfram-minut-full.nb, Elementy-jezyka-Wolfram-minut.nb,
 Elementy-jezyka-Wolfram poprawki redakcji.pdf, inne-pliki,
 rrzm_przekroj_lac.txt, str-tyt, test1.txt, test2.txt, wyn1.txt, wyn2.txt}
```

Wyświetlmy zawartość interesującego nas pliku.

In[459]:=
```
FilePrint["rrzm_przekroj_lac.txt"]
```

```
Z coordinate of the cutting plane = 9770
41
1 4396 4248 310
2 5509 3548 300
3 8489 6588 350
4 9377 1071 210
5 8865 548 310
6 3233 7672 120
7 6653 8675 290
8 6600 7762 280
9 1371 2859 350
10 5091 9229 320
11 7661 8904 350
12 2765 1340 340
13 2215 6770 120
14 3512 9843 340
15 3286 3571 320
16 1108 6834 290
17 8016 4691 320
18 3205 681 330
19 2925 4938 290
20 2536 6729 220
21 3462 6117 310
22 115 3010 320
23 1775 8530 320
24 9508 8427 240
25 3339 7020 250
26 6197 6789 300
27 4671 7777 260
28 2041 126 330
29 2212 3276 240
30 8509 7952 230
31 8407 9558 340
32 1051 124 350
33 7516 6635 240
34 7156 6110 350
35 4685 4862 280
36 8490 8769 320
37 1284 2175 350
38 2023 1244 260
39 5416 8044 320
40 4308 5897 350
41 2490 7399 310
0
0
```

Początek nazwy pliku (w praktyce do przetworzenia było wiele plików o różnych początkach nazw).

In[460]:=
```
struktura = "rrzm"
```

Out[460]=
```
rrzm
```

Wczytajmy każdą linię pliku jako osobny łańcuch tekstowy.

```
In[461]:= d = ReadList[struktura <> "_przekroj_lac.txt", String]
```

```
Out[461]= {Z coordinate of the cutting plane = 9770, 41, 1 4396 4248 310, 2 5509 3548 300,
  3 8489 6588 350, 4 9377 1071 210, 5 8865 548 310, 6 3233 7672 120, 7 6653 8675 290,
  8 6600 7762 280, 9 1371 2859 350, 10 5091 9229 320, 11 7661 8904 350, 12 2765 1340 340,
  13 2215 6770 120, 14 3512 9843 340, 15 3286 3571 320, 16 1108 6834 290, 17 8016 4691 320,
  18 3205 681 330, 19 2925 4938 290, 20 2536 6729 220, 21 3462 6117 310, 22 115 3010 320,
  23 1775 8530 320, 24 9508 8427 240, 25 3339 7020 250, 26 6197 6789 300, 27 4671 7777 260,
  28 2041 126 330, 29 2212 3276 240, 30 8509 7952 230, 31 8407 9558 340, 32 1051 124 350,
  33 7516 6635 240, 34 7156 6110 350, 35 4685 4862 280, 36 8490 8769 320, 37 1284 2175 350,
  38 2023 1244 260, 39 5416 8044 320, 40 4308 5897 350, 41 2490 7399 310, 0, 0}
```

Druga linia zawiera interesującą nas liczbę cząstek.

```
In[462]:= n = ToExpression[d[[2]]]
```

```
Out[462]= 41
```

Wydzielmy do nowej zmiennej dane poszczególnych cząstek.

```
In[463]:= dd = Table[d[[i]], {i, 3, n + 2}]
```

```
Out[463]= {1 4396 4248 310, 2 5509 3548 300, 3 8489 6588 350, 4 9377 1071 210, 5 8865 548 310,
  6 3233 7672 120, 7 6653 8675 290, 8 6600 7762 280, 9 1371 2859 350, 10 5091 9229 320,
  11 7661 8904 350, 12 2765 1340 340, 13 2215 6770 120, 14 3512 9843 340,
  15 3286 3571 320, 16 1108 6834 290, 17 8016 4691 320, 18 3205 681 330,
  19 2925 4938 290, 20 2536 6729 220, 21 3462 6117 310, 22 115 3010 320, 23 1775 8530 320,
  24 9508 8427 240, 25 3339 7020 250, 26 6197 6789 300, 27 4671 7777 260, 28 2041 126 330,
  29 2212 3276 240, 30 8509 7952 230, 31 8407 9558 340, 32 1051 124 350, 33 7516 6635 240,
  34 7156 6110 350, 35 4685 4862 280, 36 8490 8769 320, 37 1284 2175 350,
  38 2023 1244 260, 39 5416 8044 320, 40 4308 5897 350, 41 2490 7399 310}
```

Dane cząstek zawierają kolejno: numer cząstki (który nas nie interesuje), współrzędne x oraz y jej środka, a także promień cząstki. Dane te są oddzielone spacjami. Dlatego odczytamy położenie spacji w łańcuchu.

```
In[464]:= m = StringPosition[dd[[1]], " "]
```

```
Out[464]= {{2, 2}, {7, 7}, {12, 12}}
```

Możemy już teraz wypisać współrzędne środka cząstki (dalej jako łańcuchy tekstowe).

```
In[465]:= {StringTake[dd[[1]], {m[[1, 1]] + 1, m[[2, 1]] - 1}],
           StringTake[dd[[1]], {m[[2, 1]] + 1, m[[3, 1]] - 1}]}
```

```
Out[465]= {4396, 4248}
```

A teraz promień cząstki.

```
In[466]:= StringTake[dd[[1]], {m[[3, 1]] + 1, StringLength[dd[[1]]]}]
```

```
Out[466]= 310
```

W celu automatyzacji możemy zebrać powyższe instrukcje w procedurę.

In[467]:=
```
Clear[daneZpliku];
daneZpliku[plik_] := Module[{d, n, m, dd, xx1, xp1, xx, xp, i},
   d = ReadList[plik, String];
   n = ToExpression[d[[2]]];
   dd = Table[d[[i]], {i, 3, n + 2}];
   xx1 = {};
   xp1 = {};

   Do[m = StringPosition[dd[[i]], " "];
    AppendTo[xx1, {StringTake[dd[[i]], {m[[1, 1]] + 1, m[[2, 1]] - 1}],
      StringTake[dd[[i]], {m[[2, 1]] + 1, m[[3, 1]] - 1}]}];
    AppendTo[xp1, StringTake[dd[[i]], {m[[3, 1]] + 1, StringLength[dd[[i]]]}]],
    {i, 1, n}];

   xx = ToExpression[xx1];
   xp = ToExpression[xp1];

   Return[{xx, xp}]];
```

Uruchamiając procedurę, dostaniemy położenie środków oraz promienie cząstek (już jako liczby).

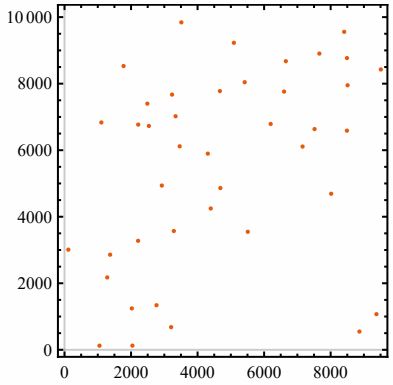
In[469]:=
```
{xx, xp} = daneZpliku[struktura <> "_przekroj_lac.txt"]
```

Out[469]=
```
{{{4396, 4248}, {5509, 3548}, {8489, 6588}, {9377, 1071}, {8865, 548},
  {3233, 7672}, {6653, 8675}, {6600, 7762}, {1371, 2859}, {5091, 9229}, {7661, 8904},
  {2765, 1340}, {2215, 6770}, {3512, 9843}, {3286, 3571}, {1108, 6834}, {8016, 4691},
  {3205, 681}, {2925, 4938}, {2536, 6729}, {3462, 6117}, {115, 3010}, {1775, 8530},
  {9508, 8427}, {3339, 7020}, {6197, 6789}, {4671, 7777}, {2041, 126}, {2212, 3276},
  {8509, 7952}, {8407, 9558}, {1051, 124}, {7516, 6635}, {7156, 6110}, {4685, 4862},
  {8490, 8769}, {1284, 2175}, {2023, 1244}, {5416, 8044}, {4308, 5897}, {2490, 7399}},
 {310, 300, 350, 210, 310, 120, 290, 280, 350, 320, 350, 340, 120, 340,
  320, 290, 320, 330, 290, 220, 310, 320, 320, 240, 250, 300, 260, 330,
  240, 230, 340, 350, 240, 350, 280, 320, 350, 260, 320, 350, 310}}
```

Możemy teraz wykreślić położenie środków cząstek.

In[470]:=
```
p1 = ListPlot[xx, Frame → True,
   PlotTheme → "Scientific", AspectRatio → Automatic, ImageSize → 200]
```

Out[470]=

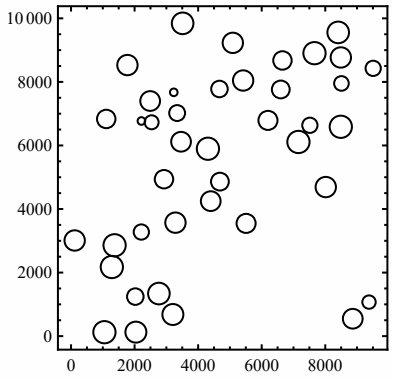


A także wykreślić same cząstki.

In[471]:=
```
okregi = Table[Circle[xx[[i]], xp[[i]]], {i, 1, Length[xx]}];
```

In[472]:=
```
p2 = Show[Graphics[okregi], AspectRatio → Automatic, Frame → True, ImageSize → 200]
```

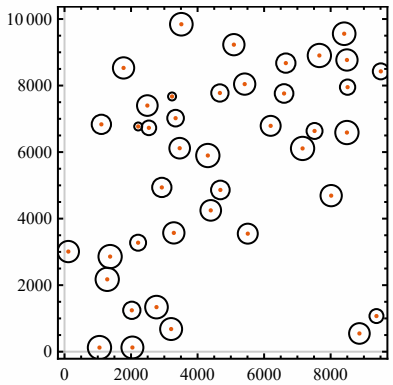
Out[472]=



Połączmy ostatnie dwa rysunki.

In[473]:=
```
Show[p1, p2, ImageSize → 200]
```

Out[473]=



3.3. Eksport i import grafiki

In[474]:=
```
Clear["Global`*"]
```

Jeśli w programie *Mathematica* tworzymy grafikę i korzystamy przy tym z jej interfejsu *FrontEnd*, możemy do przesyłania grafiki do innego programu użyć znanego mechanizmu kopiuj - wklej. Aby skopiować grafikę do schowka lub zapisać ją na dysku w wybranym formacie, stosować też możemy polecenia *Copy As* z menu *Edit*, *Save Selection As...* umieszczone w menu *File* lub Save Graphics As... z menu rozwijanego po kliknięciu na grafice prawym klawiszem myszy. Jednak znacznie większe możliwości daje nam instrukcja Export. Z kolei importować grafikę do programu możemy wykorzystując polecenie *Paste* z menu *Edit*, jeśli grafikę mamy w schowku, lub instrukcję Import, gdy mamy ją zapisaną na dysku.

```
Export[''nazwa.rozszerzenie'', grafika]  eksportuje grafikę do podanego pliku w formacie określonym przez rozszerzenie;
 Export[''plik'', grafika, ''format'']          eksportuje grafikę do podanego pliku we wskazanym formacie;
    Import[''nazwa.rozszerzenie'']      importuje grafikę z podanego pliku w formacie określonym przez rozszerzenie;
     Import[''plik'', ''format'']               importuje grafikę z podanego pliku we wskazanym formacie.
```

Obsługiwane formaty zapisane są w zmiennych systemowych $ExportFormats i $ImportFormats.

Przy eksporcie grafiki musimy czasami podać wymiary, jakie ma mieć grafika. Wymiary te możemy określić, używając opcji ImageSize. Standardowo tworzony jest obraz o szerokości 4 cali.

```
   ImageSize → x     określa szerokość grafiki jako x punktów drukarskich;
 ImageSize → 72 * xi         określa szerokość grafiki równą xi cali;
ImageResolution → r     pozwala określić rozdzielczość obrazu (r dpi).
```

Przy imporcie grafiki pliki są poszukiwane tylko w katalogach będących w ścieżce wyszukiwania zbiorów programu *Mathematica*. Informacja o katalogach zawartych w tej ścieżce jest zapisana w zmiennej $Path. Jeśli wczytywany plik nie znajduje się w żadnym

katalogu umieszczonym w tej ścieżce, wówczas w wywołaniu instrukcji Import musimy podać pełną ścieżkę dostępu do wczytywanego pliku lub instrukcją SetDirectory ustawić aktualny katalog roboczy na katalog zawierający wczytywany plik. Także instrukcja Export zapisuje tworzoną grafikę w aktualnym katalogu roboczym lub w miejscu wskazanym przez podaną w jej wywołaniu ścieżkę dostępu.

```
          $Path              podaje ścieżkę wyszukiwania zbiorów;
       Directory[]             podaje aktualny katalog roboczy;
SetDirectory[''katalog'']    ustawia aktualny katalog roboczy;
   NotebookDirectory[]       podaje katalog bieżącego notatnika.
```

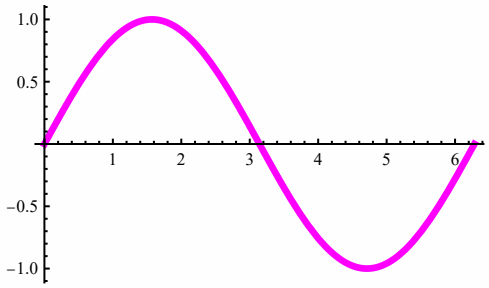
Zmieńmy aktualny katalog roboczy.

```
In[475]:= SetDirectory[NotebookDirectory[]]
Out[475]= D:\Damian\zajecia\szkolenie-f2s\minut-2
```

Narysujmy wykres.

```
In[476]:= rys = Plot[Sin[x], {x, 0, 2 π},
              PlotStyle → {RGBColor[1, 0, 1], Thickness[0.015]}, ImageSize → 250]
```

Out[476]=



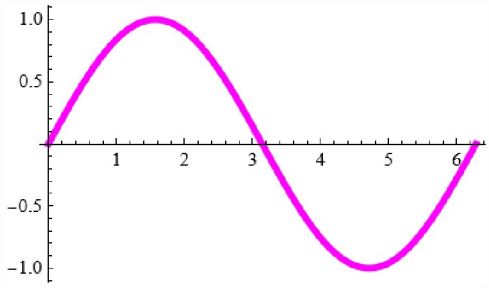
Możemy go teraz zapisać na dysku w aktualnym katalogu roboczym.

```
In[477]:= Export["sinus.png", rys]
Out[477]= sinus.png
```

Wczytajmy zapisany rysunek.

```
In[478]:= e = Import["sinus.png"]
```

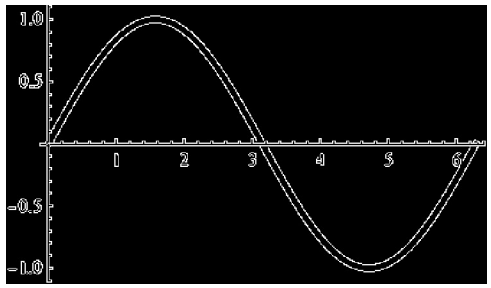
Out[478]=



Obrazki możemy poddawać różnym przekształceniom.

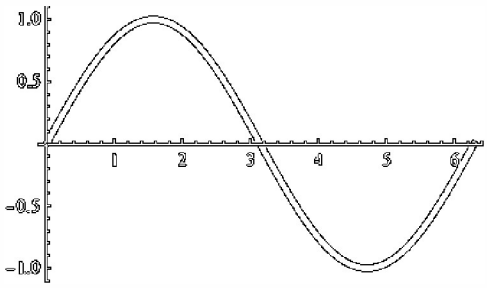
```
In[479]:= EdgeDetect[e]
```

Out[479]=



In[480]:=
```
ColorNegate[EdgeDetect[e]]
```

Out[480]=



Możemy także wczytywać do programu zdjęcia oraz rysunki.

In[481]:=
```
aa = Import["20240717_135007.jpg"];
```

Wyświetlmy informacje o wczytanym zdjęciu.

In[482]:=
```
Information[aa]
```

Out[482]=

Image	
Color Space	RGB
Image Dimensions	{4000, 3000}
Image Resolution	Automatic
Channels	3
Interleaving	True
Transparency	False
Data Type	Byte

Zmieńmy wymiary zdjęcia.

In[483]:=
```
aa1 = ImageResize[aa, 250]
```

Out[483]=

In[484]:=
```
Information[aa1]
```

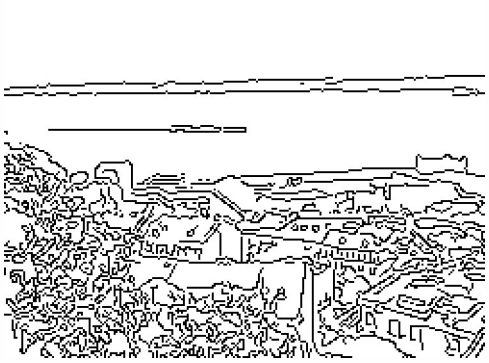
Out[484]=

Image	
Color Space	RGB
Image Dimensions	{250, 188}
Image Resolution	Automatic
Channels	3
Interleaving	True
Transparency	False
Data Type	Byte

Wczytane zdjęcie możemy poddawać różnym przekształceniom.

In[485]:=
```
ColorNegate[EdgeDetect[aa1]]
```

Out[485]=



In[486]:=
```
Blur[aa1, 5]
```

Out[486]=