Płocka Akademia Olimpijczyka

Podręcznik do programowania

Mariusz Różycki

Wersja z dnia 28 września 2019

Spis treści

Wprowadzenie

Poniższy dokument stanowi podręcznik programowania dla grupy informatycznej Płockiej Akademii Olimpijczyka. Jest on w trakcie tworzenia, stąd niektóre rozdziały mogą być niekompletne lub całkowicie puste.

Język programowania

Podręcznik ten omawia podstawowe zagadnienia związane z programowaniem na podstawie języka Python przy użyciu wbudowanego modułu `turtle`.

1. PyCharm Edu

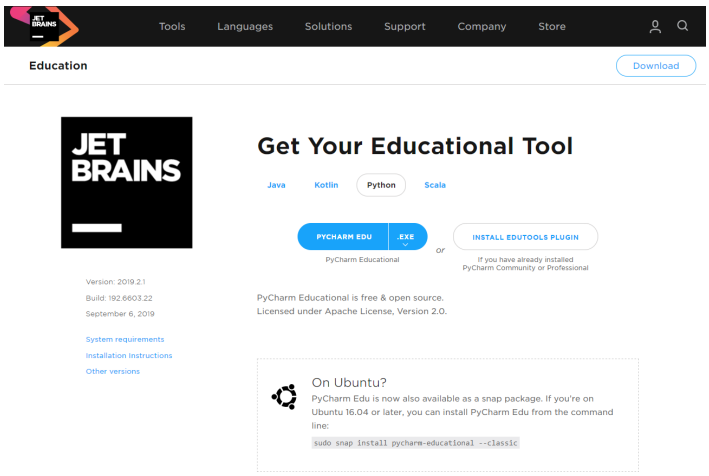
Do nauki Pythona używać będziemy programu PyCharm Edu. Jest to tak zwane zintegrowanie środowisko programistyczne, (po angielsku Integrated Development Environment, w skrócie IDE). Oznacza to, że zawiera nie tylko edytor tekstu ułatwiający pisanie kodu, ale także narzędzia pozwalające na jego uruchamianie testowanie i ulepszanie.

Program ten jest całkowicie darmowy i dostępny w wersjach na systemy Linux, Windows i macOS, co ułatwi wam ćwiczenie programowania poza zajęciami, w domu.

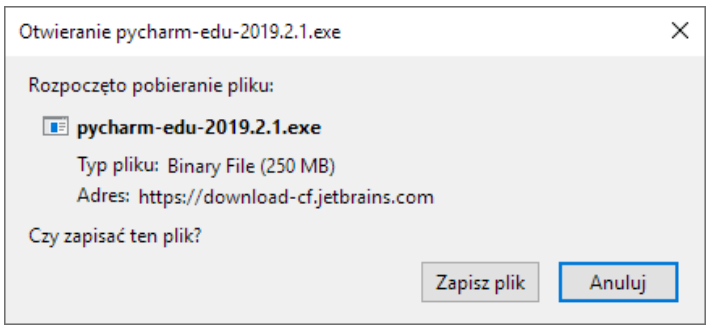
W tym rozdziale dowiemy się jak pobrać i zainstalować program PyCharm Edu w systemie Windows oraz jak zacząć z nim pracę.

Pobranie i instalacja programu

Program PyCharm Edu można pobrać za darmo ze strony jego producenta, JetBrains. Wchodzimy pod adres `https://www.jetbrains.com/education/download/#section=pycharm-edu` i klikamy przycisk PYCHARM EDU.

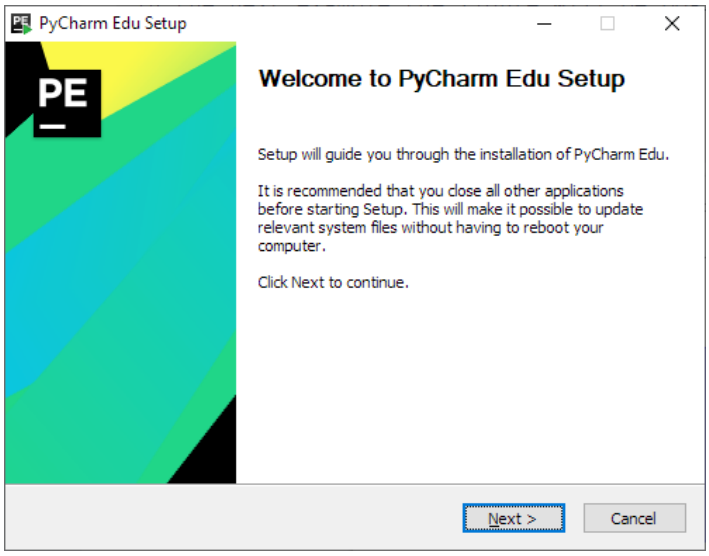


Po chwili powinno pojawić się okienko podobne do poniższego, proszące o potwierdzenie pobrania pliku. Dokładny jego wygląd może różnić się w zależności od używanej przeglądarki internetowej. Klikamy Zapisz plik.

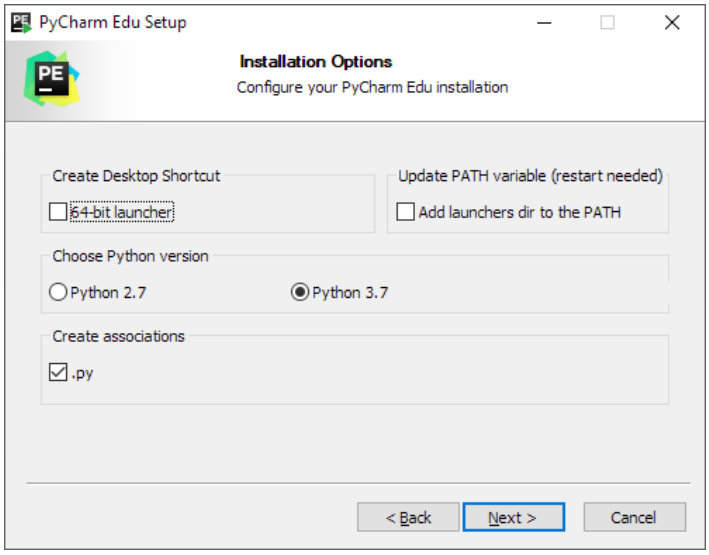


Kiedy pobieranie zakończy się, możemy uruchomić instalator. Jeżeli po-

jawi się ostrzeżenie o uruchamianiu programów pobranych z internetu, możemy je bezpiecznie zignorować – wiemy, że ten program jest bezpieczny. Pozwalamy też na wprowadzanie programowi zmian w systemie, ponieważ bez tego nie da się nic zainstalować.



Instalacja programu PyCharm Edu przypomina instalację wszystkich innych programów w systemie Windows i wymaga kilkukrotnego kliknięcia przycisku Next (Dalej). Jedną z rzeczy na którą musimy zwrócić uwagę jest instalowana wersja Pythona. Domyślnie powinna być wybrana opcja Python 3.7i to właśnie tę chcemy zainstalować.

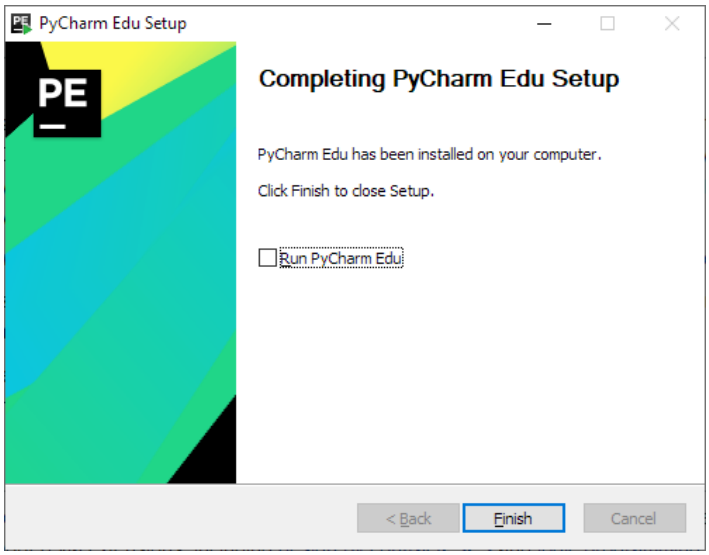


Kiedy pojawi nam się okno jak powyżej, upewniamy się, że zaznaczona jest opcja Python 3.7. Możliwe też, że taka wersja jest już zainstalowana w systemie, wtedy żadna z opcji nie będzie zaznaczona, a obok wersji Python 3.7 pojawi się napis Installed.

Więcej informacji…

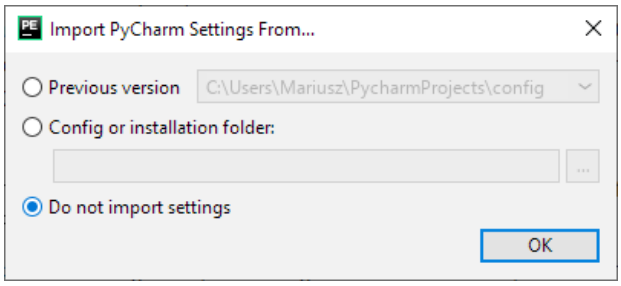
Rozbicie Pythona na wersje 2 i 3 jest najlepszym przykładem na to, że wszystkie języki programowania i narzędzia tworzone są przez ludzi, którzy nie są nieomylni. Wersja 3 jest zdecydowanym ulepszeniem języka, jednak nie jest wstecznie kompatybilna z wersją 2. Oznacza to, że programy napisane w wersji 2 nie zawsze będą działać w wersji 3 i vice versa. Zmiana z wersji 2 na 3 okazała się zbyt kosztowna dla wielu użytkowników Pythona i stąd starsza wersja wciąż jest używana przez wiele osób i wspierana przez jego twórców – mimo tego że wersja 3 istnieje już od ponad 10 lat!

Po kilku kolejnych kliknięciach przycisku Next i w końcu Install, będziemy musieli zaczekać kilka minut aż instalator pobierze i zainstaluje odpowiednią wersję Pythona, a następnie właściwy PyCharm Edu. Po zakończeniu tego procesu pojawi nam się okno zakończenia instalacji, w którym możemy kliknąć Finish. Jeżeli chcemy od razu uruchomić program, możemy najpierw zaznaczyć opcję Run PyCharm Edu. Później program możemy uruchomić odnajdując go w menu Start, pod Wszystkie Programy, JetBrains.

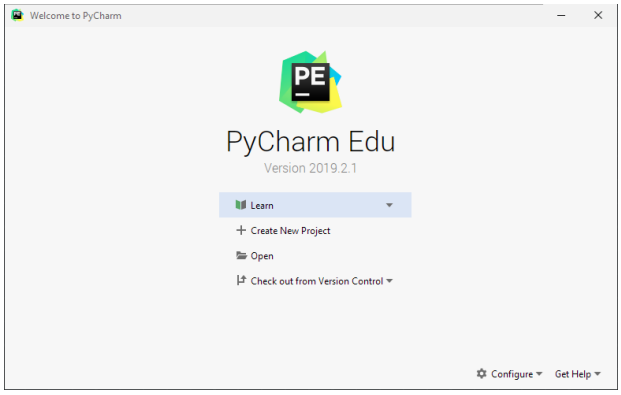


Pierwsze uruchomienie programu

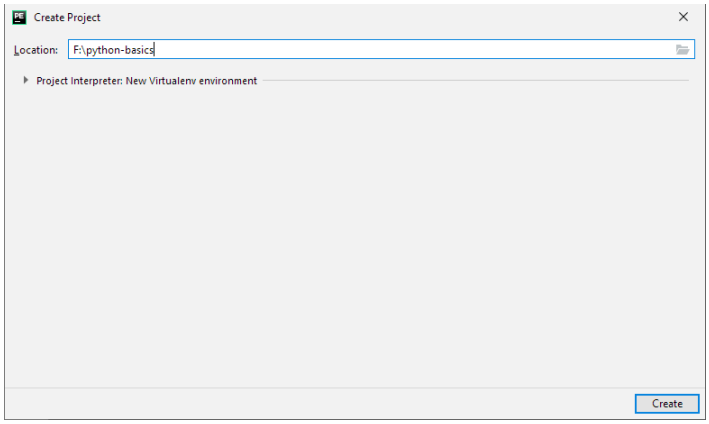
Przy pierwszym uruchomieniu programu zobaczmy poniższe okienko, które pozwoli nam na import ustawień z poprzedniej wersji. Jako że jej nie posiadamy, zostawiamy domyślną opcję Do not import settings i klikamy OK.



Zostaniemy też poproszeni o zaakceptowanie umowy licencyjnej. Następnie zobaczymy okno powitalne, takie jak poniżej.



Aby zacząć programować, musimy stworzyć nowy projekt. Klikamy w tym celu przycisk Create New Project. Zostaniemy poproszeni o podanie lokalizacji dla nowego projektu. Możemy wybrać inną niż sugerowana używając przycisku folderu znajdującego się po prawej stronie pola tekstowego. Następnie klikamy Create, aby zakończyć tworzenie projektu.

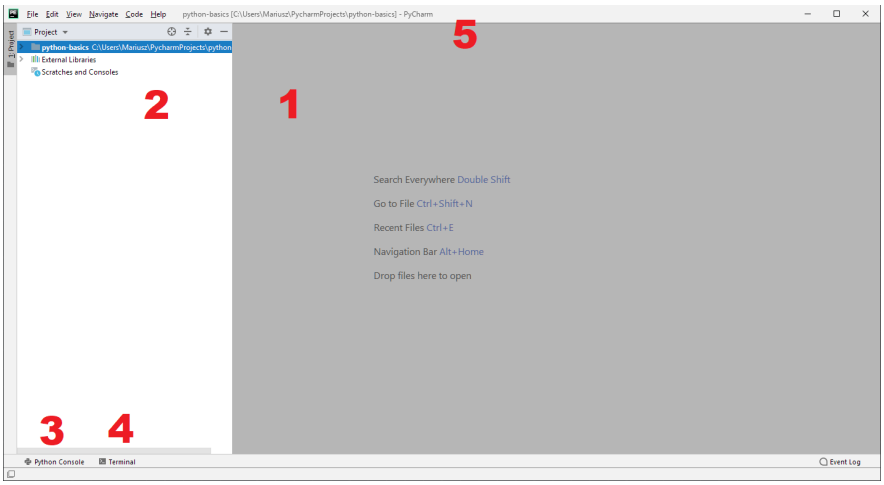


Uwaga!

Ważne jest, żeby zapamiętać gdzie został utworzony projekt. Mimo że PyCharm Edu domyślnie otwiera po starcie ostatnio używany projekt, to na współdzielonych komputerach w pracowni nie musi to być koniecznie twój projekt.

Zalecamy przynoszenie na zajęcia własnego pendrive'a i zapisywanie na nich swojego projektu. Umożliwi to też pracę nad tymi samymi programami w domu. Urządzenia o pojemności 16GB będą więcej niż wystarczające, a obecnie można je kupić nawet za 20zł.

Po kilku chwilach, zobaczymy puste okno projektu. Składa się ono z dwóch głównych części: edytora (1), obecnie pustej szarej przestrzeni, gdzie będziemy pisać nasze programy, oraz widoku projektu (2), gdzie widzimy pliki należące do naszego projektu. Dowiemy się o nich nieco więcej w kolejnych rozdziałach.



Na dole ekranu znajdują się również przyciski aktywujące dwa ważne narzędzia: konsolę Pythona (3), z której będziemy korzystać w kolejnym rozdziale, oraz terminal (4), który pozwala na interakcję z komputerem poprzez tekstowy interfejs. Przyda się on nam w późniejszych rozdziałach.

Okno programu zawiera również pasek tytułowy i górne menu (5), z którego będziemy mogli się później dostać do bardziej zaawansowanych funkcji programu.

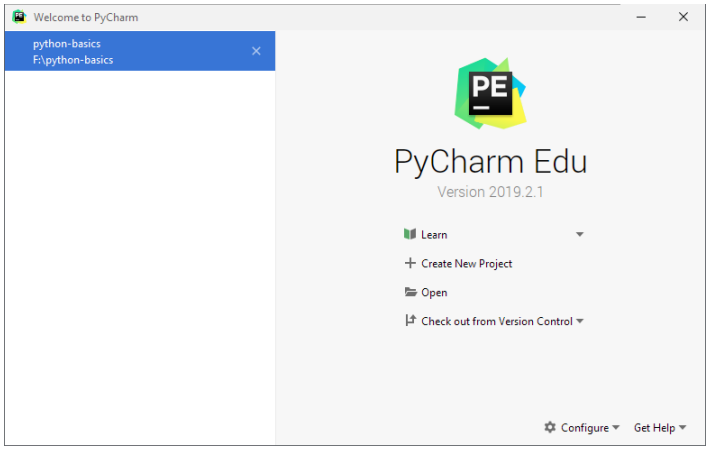
O tym jak programować przy użyciu PyCharm Edu dowiemy się więcej w kolejnych rozdziałach.

Dalsza praca z programem

Przy kolejnych uruchomieniach PyCharm Edu nie pojawi nam się już okno powitalne. Program domyślnie załaduje ostatnio używany projekt. Jednak nadal przyda się nam wiedza jak otworzyć wcześniej utworzony projekt. Można to zrobić na kilka sposobów.

Ostatnio używane projekty na ekranie powitalnym

Może zdarzyć się, że przy kolejnym uruchomieniu programu, zamiast naszego projektu zobaczymy znów ekran powitalny. Tym razem jednak będzie on zawierać listę ostatnio używanych projektów. Wystarczy kliknąć odpowiednią pozycję, aby otworzyć projekt.



Uwaga!

Jeżeli utworzysz swój projekt na pendrivie, to na liście ostatnio używanych może być on podświetlony na czerwono. Stanie się tak, jeżeli pendrive z projektem nie będzie podłączony do komputera. Mimo że PyCharm Edu pamięta gdzie znajdował się twój projekt, to nie może się do niego dostać. Aby rozwiązać ten problem, podłącz do komputera pendrive ze swoim projektem.

Przycisk Open

Jeżeli uruchamiasz swój projekt z pendrive'a po raz pierwszy na nowym komputerze (na przykład w domu po zajęciach), to twojego projektu nie będzie na liście ostatnio używanych. Aby go otworzyć, musisz kliknąć przycisk Open na ekranie powitalnym i wskazać gdzie projekt się znajduje. Pamiętaj, żeby podłączyć najpierw swój pendrive!

Zamykanie otwartego projektu

Jeżeli program uruchomi się i załaduje projekt, który nie jest twój, możesz go zamknąć i otworzyć inny projekt. Wybierz z górnego menu File, Close Project. Pojawi się wtedy okno powitalne, które pozwoli ci otworzyć swój projekt zgodnie z instrukcjami powyżej.

Podsumowanie

- PyCharm Edu to zintegrowane środowisko programistyczne (po angielsku, w skrócie IDE). Oznacza to, że nie tylko umożliwia nam pisanie kodu, ale także zawiera narzędzia które ułatwiają jego uruchamianie, testowanie i poprawianie.

- Program ten jest darmowy i dostępny na różne systemy operacyjne. Możemy go pobrać ze strony producenta – JetBrains.
- Aby zacząć korzystanie z programu, musimy utworzyć nowy projekt. Zawierać on będzie programy napisane w czasie tego kursu.
- Przy kolejnym uruchomieniu programu, nasz projekt powinien otworzyć się automatycznie. Gdyby się tak nie stało, istnieje kilka sposobów, aby otworzyć go ręcznie. Musimy jednak pamiętać, gdzie go zapisaliśmy!

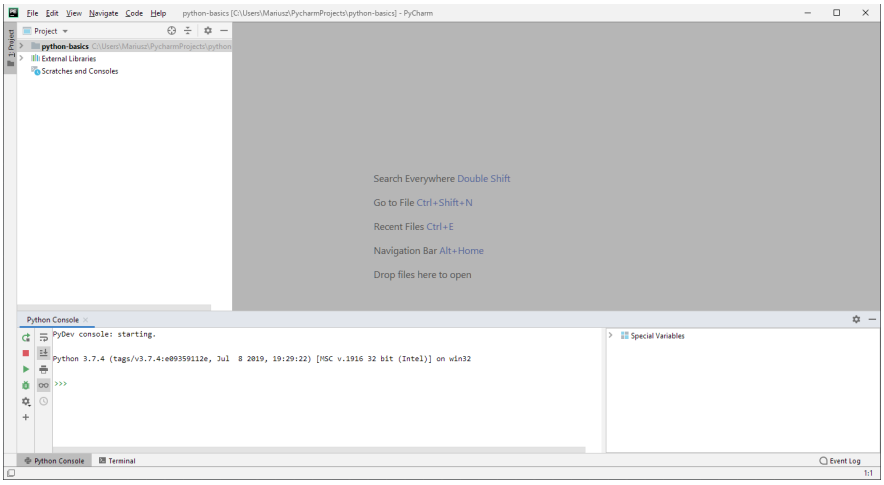
2. Poruszanie się żółwia

W tym rozdziale nauczymy się poruszać żółwiem, który zostawiać będzie za sobą ślad, a tym samym tworzyć rysunki. Nauczymy się dzięki temu posługiwać konsolą języka Python, oraz wywoływać wbudowane funkcje.

Posługiwanie się konsolą Pythona

W tym rozdziale będziemy wydawać komputerowi pojedyncze polecenia, używając konsoli Pythona. Pozwala to na szybkie zapoznanie się z funkcjami grafiki żółwia, a także eksperymentowanie, nawet w późniejszych etapach nauki. Jest to bardzo przydatne, a jednocześnie proste w użyciu narzędzie.

Aby je uruchomić, należy kliknąć przycisk Python Console znajdujący się w lewym dolnym rogu okna programu naszego środowiska. Otworzy to nowy obszar na dole ekranu.



Dla ułatwienia pracy, możemy ten obszar powiększyć, klikając na jego górną krawędź i przeciągając ją do góry.

Na górze konsoli widzimy wersję Pythona, której używamy, z dodatkowymi szczegółami, które nie są dla nas istotne. Poniżej znajdują się trzy strzałki. Jest to tak zwany znak zachęty (ang. prompt), który sygnalizuje gotowość konsoli do przyjęcia kolejnego polecenia.

Możemy wypróbować działanie konsoli używając jakiegoś prostego polecenia, na przykład działania matematycznego. W tym celu klikamy w okienko konsoli, aby umieścić kursor tuż za znakiem zachęty, wpisujemy `2+2` i wciskamy klawisz Enter.

Wynik działania, liczba 4, pojawi się pod naszym poleceniem, a pod nim kolejny znak zachęty. Po wykonaniu każdego polecenia, konsola wypisze nowy znak zachęty, informując nas tym samym, że zakończyła wykonywanie polecenia. Zazwyczaj będzie to natychmiastowe, jednak dla poleceń które wymagają trochę więcej czasu, może się to opóźnić.

Spróbuj!

Programowania najlepiej uczyć się poprzez praktykę i eksperymentowanie, więc czas poeksperymentować! Nie bój się, niczego nie zepsujesz. Spróbuj znaleźć wyniki innych działań matematycznych używając konsoli Pythona. Jakie inne operacje można wykonywać? Jaka jest kolejność działań? Czy można używać nawiasów? Co stanie się, jeżeli wpiszesz coś co nie jest działaniem matematycznym?

Komputery są bardzo wrażliwe na często niewielkie pomyłki w programach. Jeżeli zrobisz literówkę, albo użyjesz nie tego znaku, otrzymasz błąd. Konsola pokaże ci miejsce w którym znajduje się błąd oraz powie jaki był to rodzaj błędu, niestety po angielsku.

Na początku najbardziej narażeni będziecie na trzy rodzaje błędów:

- SyntaxError, po polsku błąd składni, zazwyczaj oznaczający użycie błędnego symbolu,
- NameError, po polsku błąd nazwy, który najczęściej oznaczać będzie literówkę w nazwie funkcji,
- TypeError, po polsku błąd typu, który zazwyczaj oznaczać będzie, że dana funkcja oczekuje innych parametrów niż te podane.

Więcej informacji…

Mimo że twórcy języków programowania dokładają wszelkich starań, żeby komunikaty o błędach były jak najbardziej zrozumiałe, to nie zawsze mogą być one oczywiste, w szczególności dla osób które dopiero uczą się programować. Wynika to z faktu, że błędy wynikające z pomyłek programisty nie zawsze występują dokładnie tam, gdzie pojawił się błąd.

Rozumienie i rozwiązywanie przyczyn błędów jest jedną z umiejętności, które przychodzą z praktyką i są codziennością dla doświadczonych programistów. Popełnianie błędów jest ludzkie – nie zniechęcajcie się więc pomyłkami. Nawet najlepsi programiści popełniają błędy.

Wywołujemy żółwia z lasu

Aby zacząć korzystać z grafiki żółwia, musimy zaimportować odpowiedni moduł, czyli powiedzieć komputerowi aby załadował odpowiednie polecenia – inaczej nie będzie wiedzieć co próbujemy zrobić.

Służy do tego polecenie `import`, po którym umieszczamy spację i nazwę modułu, w naszym przypadku `turtle`. Wpisujemy więc `import turtle` po znaku zachęty i wciskamy Enter.

Więcej informacji...

Kiedy wpiszesz słowo `import`, zostanie ono pogrubione i zmieni kolor na granatowy. Dzieje się tak za sprawą podświetlania składni, funkcji obecnej we wszystkich środowiskach programistycznych i edytorach tekstu przeznaczonych dla programistów. Środowisko będzie zmieniać wygląd pewnych słów kluczowych, po ułatwia wzrokowe poruszanie się po bardziej skomplikowanych programach, a czasem także pozwala na zauważenie pomyłek – jeżeli zrobisz literówkę w słowie `import`, to jego podświetlenie zniknie.

Po zaimportowaniu modułu `turtle`, wydaje się, że nic się nie wydarzyło. Pojawi się nam jedynie kolejny znak zachęty. Od teraz jednak konsola będzie rozumieć polecenia żółwia. Okno, po którym porusza się żółw pojawi się na ekranie po wydaniu mu pierwszego polecenia, na przykład:

```
1 turtle.showturtle()
```

Polecenie to służy do... pokazania żołwia. Po wciśnięciu Enter, powinno otworzyć się Okno z żółwiem. Może ono jednak otworzyć się w tle i nie być od razu widoczne, ale możemy je wyświetlić klikając jego ikonę w systemowym pasku zadań.

Uwaga!

W systmie Windows, okienko żołwia ma tendencję do zacinania się. System będzie myślał, że okno nie odpowiada. Mimo to kolejne polecenia będą wykonywane i mogą odwiesić okno.

Więcej informacji...

Zwróć uwagę na parę pustych nawiasów na końcu powyższego polecenia. Jest ona wymagana, ponieważ `showturtle` jest funkcją, którą musimy wywołać, poprzez dodanie do niej nawiasów. Więcej o funkcjach dowiemy się w kolejnych rozdziałach.

Pierwsze kroki

Czas zacząć poruszać żołwiem, który nie za bardzo wygląda jak żółw, a jak mała strzałka. Nasz żółw jest skierowany w stronę wskazywaną przez strzałkę i możemy mu kazać przejść do przodu. Służy do tego polecenie `forward`.

```
1 turtle.forward(100)
```

Tym razem pomiędzy nawiasami umieszczamy liczbę, która mówi ile kroków ma wykonać nasz żółw. Krok ma wielkość jednego piksela, czyli najmniejszego punktu jaki można narysować na ekranie. Po wpisaniu tego polecenia i wciśnięciu Enter, żółw przesunie się w prawo, zostawiając za sobą ślad – w ten sposób będziemy tworzyć rysunki.

Spróbuj!

Rozmiar okna po którym porusza się żółw jest zależny od kilku czynników. Sprawdź jak daleko możesz wysłać żółwia, aby zniknął z ekranu. Jaka jest szerokość okna żółwia, jeżeli żółw początkowo znajdował się na samym jego środku?

Żółw może też poruszać się do tyłu, przy pomocy polecenia `backward`.

```
1 turtle.backward(100)
```

Odległość do pononania przez żółwia nie musi być wyrażona w postaci liczby. Może być to dowolne wyrażenie arytmetyczne. Jeżeli zatem zajdzie potrzeba obliczenia długości odcinka, który będziemy chcieli narysować, to nie musimy wykonywać ich ręcznie, a możemy umieścić je w programie – komputer wykona je za nas.

Dodajemy drugi wymiar

Żółw poruszający się jedynie do przodu i do tyłu nie stworzy jednak zbyt ciekawych rysunków. Dlatego może się on też obracać. Służą do tego funkcje `left` i `right`.

```
1 turtle.left(90)
2 turtle.right(90)
```

Tutaj w nawiasach podajemy o jaki kąt żółw ma się obrócić. Jest to wartość w stopniach, a zatem pełny obrót to 360, odwrócenie się w drugą stronę to 180, a żeby zmienić kierunek z pionowego na poziomy (i vice versa) obracamy żółwia o 90 stopni. Inne wartości dadzą nam najróżniejsze rodzaje linii ukośnych.

Po wykonaniu jednego z powyższych poleceń, żółw nie przesunie się, ale obróci się w miejscu – strzałka zacznie wskazywać w innym kierunku. Następnym razem kiedy nakażemy mu przejść do przodu lub do tyłu, zrobi to w innym kierunku niż dotychczas, zgodnie ze strzałką.

Spróbuj!

Narysuj kilka figur geometrycznych, żeby oswoić się z poleceniami żółwia. Na przykład kwadrat albo kilka rodzajów trójkątów. Żeby rozpocząć rysowanie od nowa, możesz użyć funkcji `reset`.

```
1 turtle.reset()
```

Podnosimy żółwia

Jak do tej pory żółw pozostawiał za sobą ślad bez przerwy, co utrudnia tworzenie bardziej złożonych rysunków. Musielibyśmy często przechodzić w inne miejsce po narysowanych już liniach, aby nie zepsuć ostatecznego rezultatu – co na dodatek nie zawsze jest możliwe.

Aby rozwiązać ten problem, żółwia możemy podnieść (`up`). Przestanie wtedy zostawiać on za sobą ślad. Kiedy chcemy wrócić do rysowania, żółwia możemy opuścić (`down`), przywracając go do pierwotnego stanu.

```
turtle.up()
turtle.down()
```

Podsumowanie

- Konsola Pythona pozwala na wykonywanie pojedynczych poleceń.
- Aby używać grafiki żółwia, trzeba najpierw zaimportować jego moduł:

```
import turtle
```

- Okno żółwia pojawi się po wydaniu mu pierwszego polecenia.
- Aby żółw przeniósł się do przodu, używamy polecenia `forward`, np.

```
turtle.forward(100)
```

- Podobnie do tyłu przy pomocy polecenia `backward`:

```
turtle.forward(100)
```

- Żółwia możemy obrócić w lewo (`left`) lub w prawo (`right`), podając w nawiasach kąt obrotu w stopniach:

```
turtle.left(90)
turtle.right(90)
```

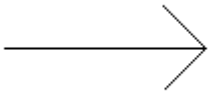
- Żółwia możemy podnieść (`up`), co spowoduje że przestanie rysować, a następnie opuścić (`down`):

```
turtle.up()
turtle.down()
```

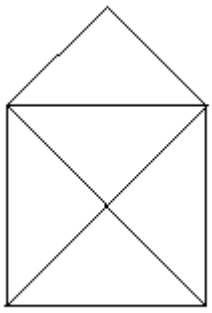
- Aby zacząć rysować od nowa, wydajemy polecenie `turtle.reset()`.

Zadania

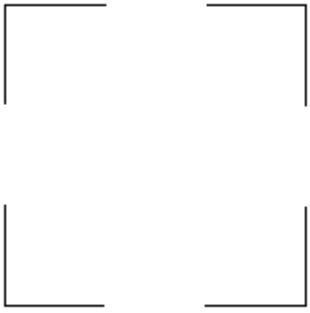
1. Narysuj kwadrat o boku 150.
2. Narysuj trójkąt równoboczny o boku 200.
3. Narysuj trójkąt równoramienny o podstawie 141 i ramieniu 100. Kąt między ramionami będzie wynosić 90 stopni.
4. Narysuj prostą strzałkę, jak na rysunku poniżej. Jej długość, oraz rozmiar i rozstaw grota możesz wybrać dowolnie, jednak rysunek powinien być symetryczny.



5. Narysuj otwartą kopertę, jak na rysunku poniżej. Postaraj się zrobić to bez podnoszenia żółwia i bez przejeżdżania dwa razy po tym samym odcinku. Zanim zaczniesz pisać, poćwicz na kartce. Podpowiedź: aby otrzymać długość przekątnej kwadratu, pomnóż jego bok przez 1.41.



6. Narysuj przerwany kwadrat jak na rysunku poniżej.



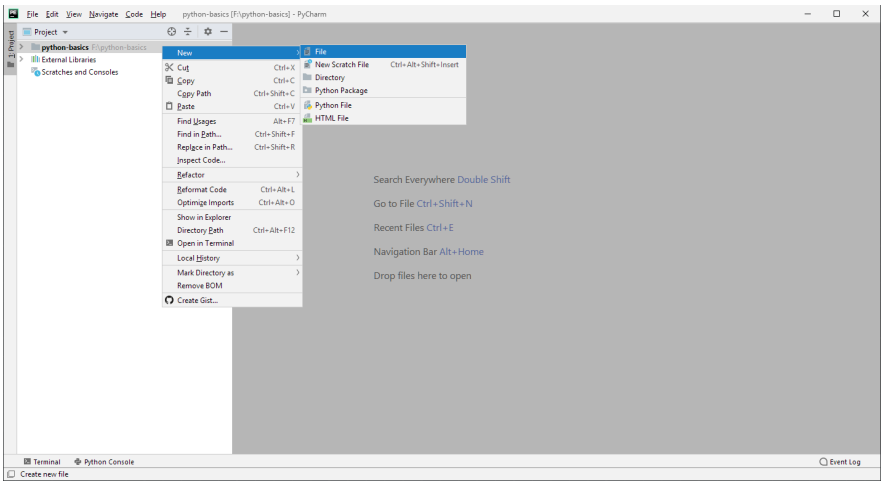
3. Zapisywanie programów w plikach

Jak zapewne zauważyliście w poprzednim rozdziale, pomimo podpowiedzi konsoli, wpisywanie wszystkich poleceń ręcznie za każdym razem jest czasochłonne. Do tego przy każdej pomyłce trzeba zazwyczaj zacząć rysowanie od nowa.

W tym rozdziale nauczymy się pisać programy w plikach i je uruchamiać. Dzięki temu będziemy mogli wielokrotnie uruchamiać ten sam program bez potrzeby jego przepisywania. Będziemy mogli także wprowadzać poprawki do istniejących programów, dzięki czemu nie będziemy musieli zaczynać pisania od nowa po każdej pomyłce.

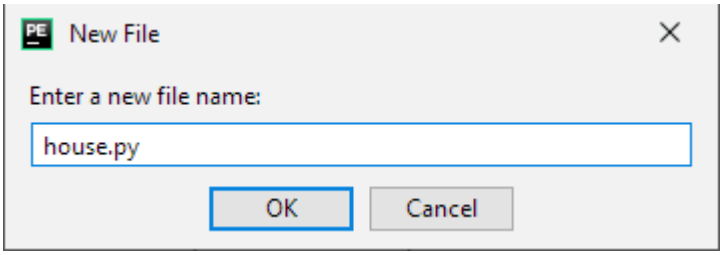
Tworzenie nowego pliku

Aby zacząć pisanie kodu w pliku, trzeba najpierw stworzyć ten plik. W tym celu w widoku projektu po lewej stronie ekranu klikamy prawym przyciskiem myszy na folder z nazwą naszego projektu i wybieramy New, File.

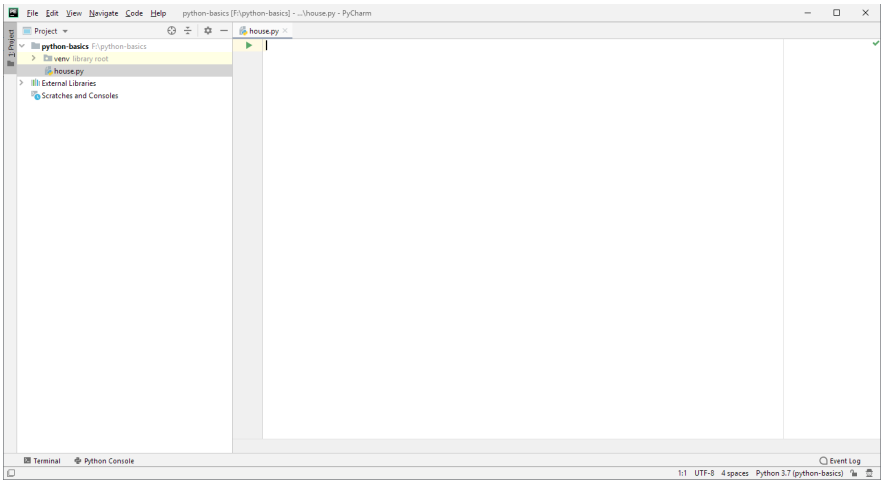


Pojawi nam się małe okienko, w którym musimy podać nazwę dla naszego pliku. Nazwa pliku może być w zasadzie dowolna, jednak nie powinna zawierać spacji ani polskich liter i musi kończyć się rozszerzeniem ".py".

Nazwa pliku powinna przede wszystkim odzwierciedlać jego zawartość. Dzięki dobrej nazwie łatwo będziemy w stanie później odnaleźć odpowiedni plik w projekcie. W tym rozdziale będziemy rysować domek, więc odpowiednią nazwą będzie `domek.py` lub po angielsku `house.py`. Wpisujemy nazwę i klikamy OK.

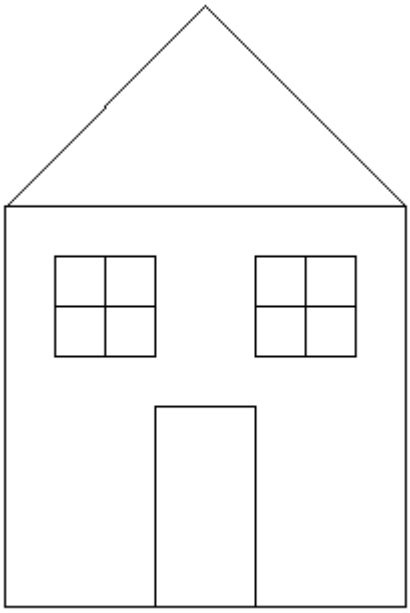


Po stworzeniu pierwszego pliku wygląd naszego środowiska zmieni się znacząco. W widoku projektu pojawi się nasz plik, a duży szary obszar po prawej stronie zostanie zastąpiony białym oknem edytora. Czas zacząć pisać!



Pierwszy program w pliku

W tym rozdziale rysować będziemy proste domki, podobne do tego poniżej:

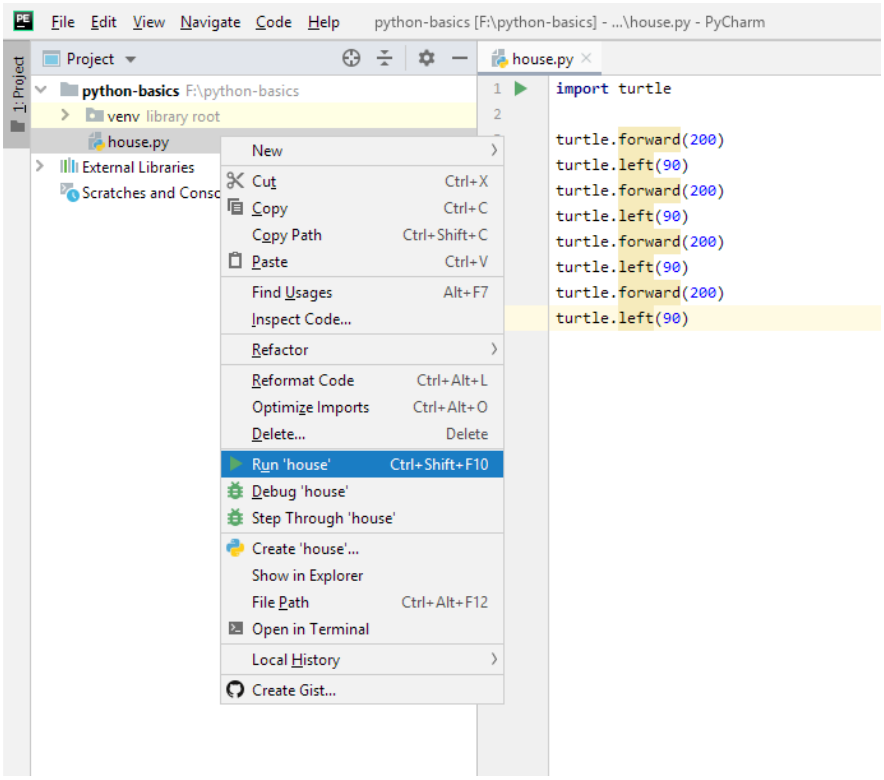


Pierwszym krokiem będzie narysowanie ściany domku, czyli kwadratu. W edytorze musimy zatem wpisać kod, który ten kwadrat narysuje. Polecenia umieszczamy jedno po drugim. Zostaną one wykonane w takiej kolejności, w jakiej umieścimy je w pliku.

```
import turtle

turtle.forward(200)
turtle.left(90)
turtle.forward(200)
turtle.left(90)
turtle.forward(200)
turtle.left(90)
turtle.forward(200)
turtle.left(90)
```

Zwróć uwagę, że w naszym programie musimy również zaimportować moduł `turtle`. Pusta linia po imporcie nie jest niezbędna, jednak pozwala wizualnie oddzielić „wymagane formalności” od właściwego kodu, co jest przydatne.

Teraz aby uruchomić nasz program, w widoku projektu prawym przyciskiem klikamy na plik z naszym kodem i wybieramy Run. Możemy też nacisnąć przycisk z zieloną strzałką na górze edytora.



Gdy to zrobimy, na ekranie pojawi się okno, na którym żółw narysuje kwadrat i… okno natychmiast zniknie. Dzieje się tak, ponieważ tuż po wykonaniu ostatniego polecenia program kończy działanie, a więc zamyka także wszystkie okna, które otworzył.

Aby temu zapobiec, dodajemy na końcu polecenie `exitonclick`, dzięki któremu program zaczeka aż na nie klikniemy i dopiero wtedy zakończy działanie. Pozwoli nam to upewnić się, że rysunek został wykonany odpowiednio.

```
import turtle
```

```
2
3  turtle.forward(200)
4  turtle.left(90)
5  turtle.forward(200)
6  turtle.left(90)
7  turtle.forward(200)
8  turtle.left(90)
9  turtle.forward(200)
10 turtle.left(90)
11
12 turtle.exitonclick()
```

Spróbuj!

Dodaj do swojego domku drzwi, dach i okno. Jeżeli chcesz, dodaj także komin. Twój rysunek nie musi wyglądać dokładnie tak samo jak mój – spróbuj zrobić coś inaczej, po swojemu. Pamiętaj jednak, że końcowy rysunek nadal powinien wyglądać jak dom!

Kolejną zaletą tworzenia programów w plikach jest to, że możemy zachować je na dłużej. Polecenia wpisane do konsoli znikają natychmiast po jej zamknięciu, natomiast program w pliku zostaje zapisany w komputerze i może zostać uruchomiony ponownie później.

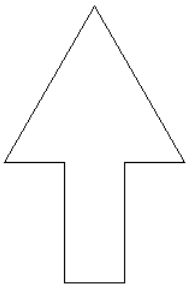
Polecam wszystkie kolejne zadania rozwiązywać w nowych plikach, dzięki temu udokumentujesz swoją naukę programowania.

Podsumowanie

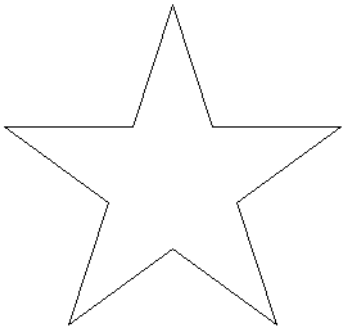
- Zamiast wpisywać kod do konsoli po jednym poleceniu, możemy wpisać cały kod do pliku.
- Aby stworzyć nowy plik, klikamy prawym przyciskiem na folder projektu w widoku projektu po lewej stronie ekranu. Wybieramy New, File, podajemy nazwę pliku i klikamy OK.
- Nazwa pliku może być dowolna, ale nie powinna zawierać spacji ani polskich liter i musi kończyć się rozszerzeniem ".py".
- Polecenia w pliku wykonywane są po kolei, od góry do dołu.
- Aby uruchomić program klikamy na jego plik prawym przyciskiem myszy i wybieramy Run.
- Możemy też kliknąć przycisk z zieloną strzałką na górze widoku edytora.
- Aby okno z żółwiem nie zamknęło się natychmiast po zakończeniu rysowania, dodajemy do programu polecenie `exitonclick`.

Zadania

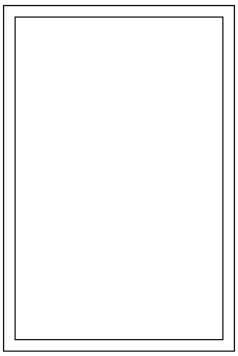
1. Stwórz plik `arrow.py` (lub `strzalka.py`) i napisz w nim program, który narysuje strzałkę taką jak poniżej:



2. Stwórz plik `star.py` (lub `gwiazda.py`) i napisz w nim program, który narysuje gwiazdę jak na rysunku poniżej. Kąt wewnątrz gwiazdy ma 36 stopni, a kąty pomiędzy ramionami po 108 stopni.



3. Stwórz plik `frame.py` (lub `ramka.py`) i napisz w nim program, który narysuje ramkę jak na rysunku poniżej.



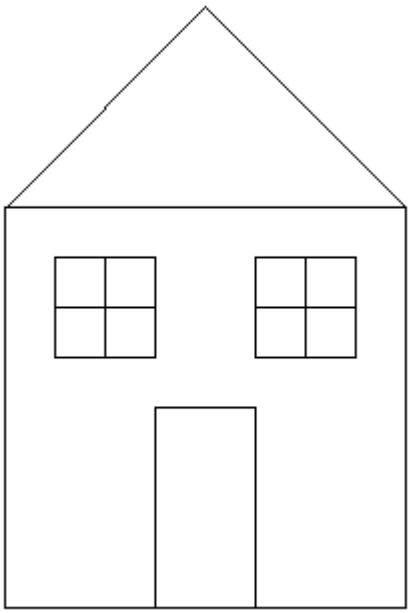
4. Procedury

Często zdarza się, że pisząc program chcemy użyć tego samego fragmentu kodu wielokrotnie. Pisząc w edytorze, możemy zawsze użyć funkcji kopiuj-wklej, jednak przy większych programach niepotrzebnie komplikuje to kod. Dodatkowo, jeżeli chcemy zmienić taki powtarzający się kod, to musimy znaleźć i poprawić wszystkie jego kopie, co może być czasochłonne, a do tego prowadzić do błędów.

W tym rozdziale nauczymy się tworzyć i używać procedur. Dzięki temu będziemy mogli takim powtarzającym się fragmentom kodu nadawać nazwy. Nie będziemy już musieli ich kopiować – kolejne użycie procedury będzie wymagało jedynie użycia jej nazwy.

Wstawiamy drugie okno

Wróćmy do rysunku domu z poprzedniego rozdziału.



Wzór podany w rozdziale miał dwa okna, jednak zadanie wymagało narysowania tylko jednego. Na pewno bez problemu moglibyśmy dodać drugie okno, kopiując odpowiedni fragment kodu w inne miejsce. Najprawdopodobniej jednak wasze domki już i tak składały się z dziesiątek poleceń i trudno było rozumieć co dokładnie się dzieje.

Zamiast kopiować kod rysujący okno, stworzymy z niego procedurę, co pozwoli nam narysować drugie okno bez kopiowania kilkunastu linii kodu w dwa miejsca. Wtedy dodanie każdego kolejnego okna będzie banalnie proste.

Wyobraźmy sobie, że nasze okno zostało narysowane tak:

```
1 turtle.forward(50)
```

```
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(25)
turtle.left(90)
turtle.forward(50)
turtle.left(90)
turtle.forward(25)
turtle.left(90)
turtle.forward(25)
turtle.left(90)
turtle.forward(50)
```

Aby utworzyć z tego kodu procedurę, musimy przenieść ten fragment kodu na początek pliku, a następnie wciąć, czyli przesunąć całość tego kodu w prawo, umieszczając na początku każdej linijki tabulator. W prostszym edytorze tekstu wymagałoby to przejścia kursorem na początek każdej kolejnej linijki i wciśnięcia przycisku Tab na klawiaturze, ale w PyCharm Edu możemy zrobić to łatwiej. Wystarczy zaznaczyć odpowiedni fragment kodu i wcisnąć Tab.

Na górze naszego wciętego fragmentu kodu musimy dodać jeszcze jedną linię, która nie tylko powie komputerowi że wcięty kod jest procedurą, ale także nada tej procedurze nazwę. Zaczyna się ona od słowa `def`, następnie spacji, nazwy procedury, pary pustych nawiasów i dwukropka. Linijka ta nie będzie wcięta, w odróżnieniu od reszty procedury.

```
def window():
    turtle.forward(50)
    turtle.left(90)
    turtle.forward(50)
    turtle.left(90)
    turtle.forward(50)
    turtle.left(90)
    turtle.forward(50)
    turtle.left(90)
    turtle.forward(25)
    turtle.left(90)
    turtle.forward(50)
    turtle.left(90)
    turtle.forward(25)
    turtle.left(90)
    turtle.forward(25)
    turtle.left(90)
    turtle.forward(50)
```

Nazwa procedury może być dowolna, podobnie jak nazwa pliku, jednak nie może zawierać spacji, nie może zaczynać się od cyfry, i nie powinna zawierać polskich znaków. Powinna być też na tyle opisowa, że po samej nazwie będziemy w stanie dowiedzieć się co robi. W powyższym przykładzie nazwaliśmy naszą procedurę `window`, od angielskiego słowa na okno, ponieważ jej zadaniem jest rysowanie właśnie okna.

Tak stworzona procedura nie zostanie jednak wykonana. Zostanie zapamiętana przez komputer, ale jej kod zostanie pominięty aż do momentu, w którym ją wywołamy, tak jak do tej pory wywoływaliśmy polecenia żółwia.

```
turtle.left(90)
turtle.forward(125)
turtle.right(90)
turtle.up()
turtle.forward(25)
turtle.down()
window()
turtle.up()
turtle.forward(100)
turtle.down()
window()
```

Uwaga!

Wywołania naszych procedur nie zaczynają się od słowa `turtle` i kropki, ponieważ nie są poleceniami żółwia. O tym dlaczego tak się dzieje dowiemy się więcej w późniejszych rozdziałach.

Więcej informacji...

Skąd komputer wie gdzie kończy się procedura, a zaczyna właściwy kod, który powinien zostać wykonany? Po wcięciach! W większości innych języków programowania wcięcia są ignorowane przez komputer i służą jedynie jako pomoc w czytaniu programu dla programisty. Python natomiast używa wcięć do decydowania gdzie zaczynają a gdzie kończą się różne fragmenty kodu, w związku z czym wymusza ich systematyczne i spójne stosowanie.

Abstrakcja

Procedury, a później także funkcje, są bardzo przydatne w unikaniu tak zwanej duplikacji kodu, czyli zamieszczania w jednym programie dwóch lub więcej kopii tego samego fragmentu kodu. Jednak nie jest to jedyna, ani nawet główna ich zaleta.

Procedury pozwalają na tworzenie abstrakcji, czyli ukrywania pewnych nieistotnych szczegółów (detali implementacyjnych), co z kolei upraszcza właściwą logikę programu. Kontynuując przykład domku, możemy stworzyć więcej metod, które będą rysować kolejne elementu domku, dzięki czemu właściwy program rysujący domek może wyglądać na przykład tak:

```
wall()

turtle.left(90)
turtle.forward(100)
roof()

turtle.left(180)
turtle.forward(25)
turtle.left(90)
turtle.up()
turtle.forward(25)
turtle.down()
window()
turtle.up()
turtle.forward(100)
turtle.down()
window()

turtle.left(180)
turtle.up()
turtle.forward(100)
turtle.left(90)
door()
```

Spróbuj!

Napisz odpowiednie procedury `wall` (ściana), `roof` (dach) i `door` (drzwi), tak aby powyższy program narysował cały domek.

W teorii, dobra abstrakcja pozwala programiście używać procedur i funkcji, nie wiedząc jak dokładnie wyglądają w środku, a wiedząc jedynie powierzchownie jakie jest ich zadanie. Dzięki temu program używający procedur, jak i same procedury, mogą być zmieniane niezależnie od siebie. Ułatwia to między innymi pracę w wieloosobowych zespołach – każda osoba może pracować nad innym fragmentem kodu. Będą one ostatecznie działać razem żeby utworzyć jeden wspólny program, jednak są od siebie na tyle niezależne, że mogą je pisać różne osoby, w różnym czasie.

Niestety w przykładzie powyżej abstrakcje utworzone przez procedury nie są idealne. Procedury rysują elementy o z góry założonych wymiarach, a co za tym idzie dodatkowy kod, który przenosi żółwia w odpowiednie miejsce musi również zawierać część informacji o tych wymiarach. Da się to naprawić, jednak zajmiemy się tym w późniejszym rozdziale.

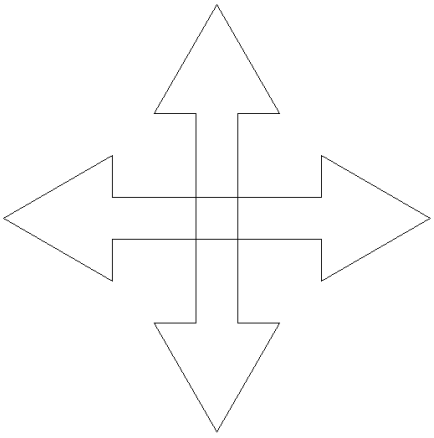
Tworzenie dobrych abstrakcji to kolejna bardzo ważna umiejętność dla programisty, której nauczenie się wymaga dużych ilości praktyki. Więcej o tworzeniu dobrych abstrakcji dowiemy się w rozdziale o funkcjach, będzie to jednak nadal tylko wierzchołek góry lodowej. Umiejętność tę będziemy ćwiczyć przez resztę tego kursu, a być może nawet kolejnych.

Podsumowanie

- Aby uniknąć duplikacji kodu, możemy powtarzający się kod zamknąć w procedurę.
- Procedurę tworzymy używając konstrukcji `def`, a następnie wcinając kod procedury o jeden poziom klawiszem Tab
- Tak stworzona procedura nie zostanie wykonana przy uruchomieniu programu, musimy ją wywołać.
- Procedurę wywołujemy poprzez podanie jej nazwy, a następnie pustej pary nawiasów.
- Procedury mogą służyć również do tworzenia abstrakcji, które ukrywają nieistotne szczegóły kodu, pozwalają na niezależne zmienianie procedury i używającego jej programu, co ułatwia pracę wielu osób nad wspólnym kodem.

Zadania

1. Posiłkując się programem rysującym strzałkę z poprzedniego rozdziału, stwórz procedurę `arrow`, która stworzy identyczny rysunek. Następnie przy jej użyciu napisz program, który stworzy następujący rysunek:



2. Procedury można wywoływać z wnętrza innych procedur. Napisz procedurę `z`, która narysuje literę Z jak na rysunku poniżej. Następnie przy jej użyciu stwórz procedurę `n`, która narysuje podobnie literę N. W końcu napisz program, który narysuje na ekranie wzór NZNZ, jak poniżej.

5. Kolory i wypełnienia

Do tej pory nasz żółw zostawiał za sobą zawsze czarny ślad, co pozwalało nam jedynie na tworzenie prostych czarno-białych rysunków.

W tym rozdziale nauczymy się zmieniać kolor śladu zostawianego przez żółwia, ale także wypełniania kształtów przez niego tworzonych. Dzięki temu nasze rysunki będą ciekawsze i bardziej oryginalne.

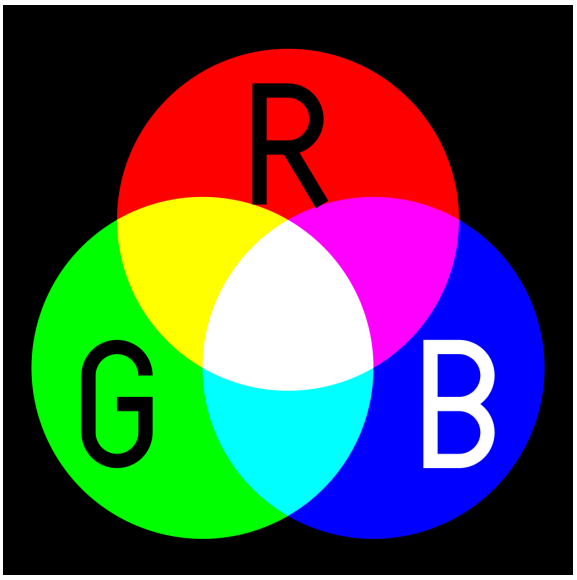
Jak ludzie widzą kolory

Zanim zaczniemy dodawać kolor do naszych rysunków, warto dowiedzieć się, chociażby w skrócie, w jaki sposób ludzie widzą kolory. Ma to znaczenie, ponieważ to w jaki sposób komputery przechowują i wyświetlają kolory jest dostosowane do tego, w jaki sposób człowiek będzie je potem potrzegać.

Ludzkie oko zawiera warstwę światłoczułą, siatkówkę, która pokryta jest dwoma rodzajami komórek: pręcikami, które odpowiadają za postrzeganie intensywności światła, oraz czopkami, które odpowiadają za postrzeganie kolorów. Tych pierwszych jest zdecydowanie więcej, około 100 milionów, zaś tych drugich jedynie 6 milionów. Na dodatek czopki dalej dzielą się na 3 rodzaje, każdy z nich reaguje najlepiej na inny kolor światła: czerwony, zielony lub niebieski.

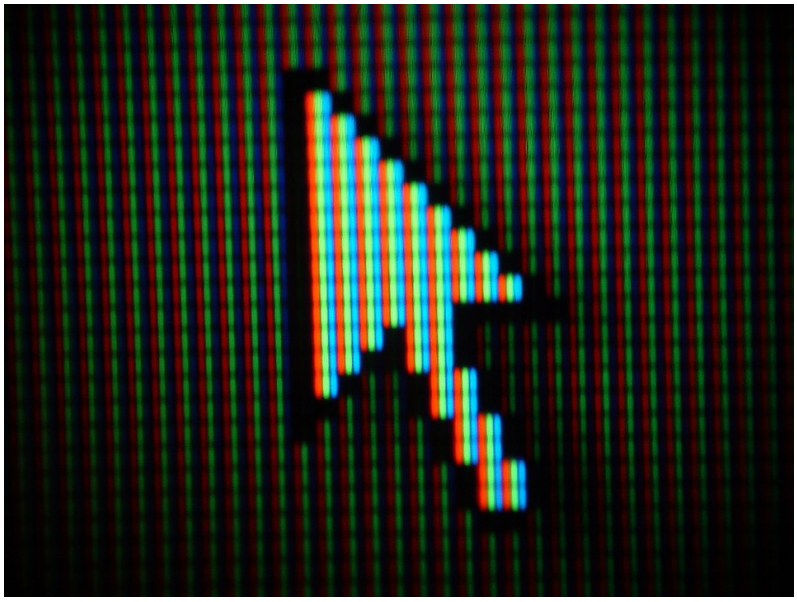
Każdy inny kolor światła wzbudza różne rodzaje czopków w różny sposób. Dzięki temu mózg jest w stanie odzwierciedlić każdy kolor tęczy (a nawet kolory, których w tęczy nie ma!) opierając się jedynie na tym jak mocny bodziec otrzymał każdy rodzaj czopków.

Dzięki temu każdy kolor widoczny dla ludzkiego oka możemy odzwierciedlić przy użyciu jedynie trzech kolorów o różnych intensywnościach: czerwonego, zielonego i niebieskiego właśnie. Nazywamy je kolorami podstawowymi.



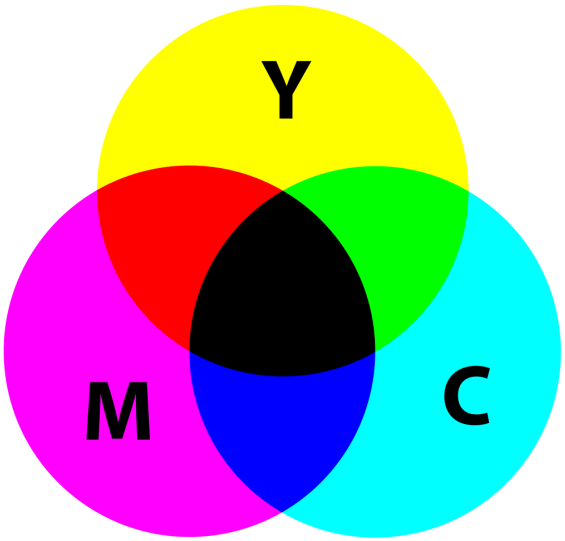
Źródło: Wikimedia Commons

Dlatego ekran komputera, czy niemalże każdego innego kolorowego wyświetlacza, składa się z milionów pikseli, czyli małych punktów, z których każdy składa się z trzech subpikseli, w trzech podstawowych kolorach. Każdy z subpikseli może świecić z różną intensywnością, tworząc różne proporcje między podstawowymi kolorami i tworząc dla ludzkiego oka złudzenie, że widzi inny kolor. Złudzenie to jednak upada, gdy przybliżymy widok monitora i zobaczymy niezależne subpiksele.



Źródło: Nevit Dilmen, Wikimedia Commons

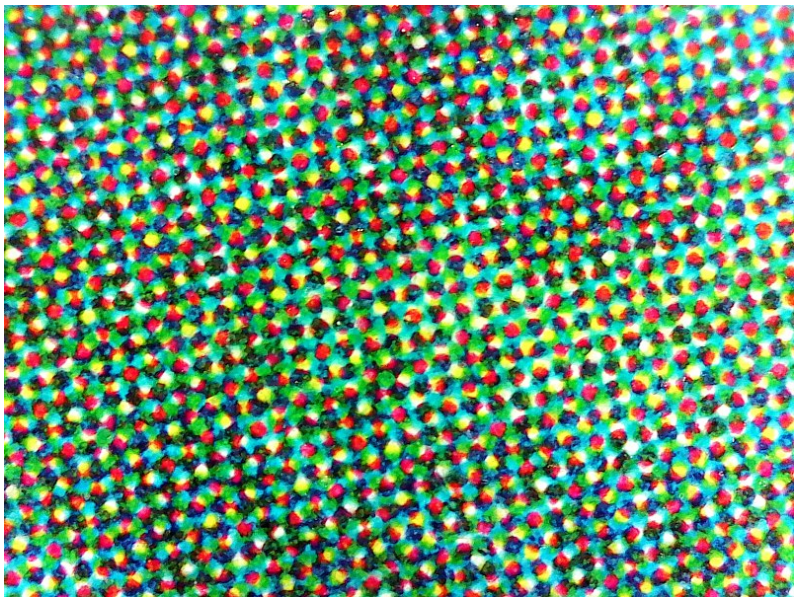
W podobny sposób możemy odzwierciedlić każdy kolor przy użyciu trzech kolorów farb: niebieskiego, czerwonego i żółtego (a w zasadzie: błękitnego, różowego i żółtego), o czym mogliście słyszeć już na lekcjach plastyki.



Źródło: Wikimedia Commons

Te trzy kolory również nazywamy podstawowymi, ale w odniesieniu do

farb. O ile podstawowe kolory światła wykorzystywane są w wyświetlaczach (które generują światło), tak podstawowe kolory farb wykorzystywane są w druku, gdzie zadrukowana powierzchnia odbija światło. Drukowane obrazy tworzone są przy pomocy małych kolorowych kropek, często z dodatkiem czarnych kropek.



Źródło: Wikimedia Commons

Komputery są w stanie przeliczać obrazy przedstawione w postaci trzech kolorów światła na obrazy w postaci 4 kolorów kropek. Czasem jednak kolory wydrukowanego obrazu mogą być błędne. Błędy takie powstają na skutek błędnej kalibracji wyświetlacza lub drukarki. Na szczęście w tym kursie nie będziemy zajmować się drukiem, więc ten problem nam nie grozi.

Kolorowe linie żółwia

Wystarczy jednak teorii, wróćmy do programowania. Do zmiany koloru linii rysowanej przez żółwia, służy polecenie `color`. Używać możemy go na kilka sposobów.

Po pierwsze, możemy w cudzysłowach podać angielską nazwę koloru, który chcemy użyć. Na przykład, aby żółw zaczął rysować na czerwono:

```
1 turtle.color("red")
```

Po wykonaniu tego polecenia, kolor żółwia zmieni się na czerwony i wszystkie narysowane przez niego linie będą teraz tego koloru. Możemy się o tym przekonać każąc żółwiowi przejść do przodu.

Więcej informacji...

Oprócz prostych barw (n.p. "blue", "yellow", "green", "black"), żółw jest w stanie rozpoznać kilkaset różnych nazw kolorów. Pełną listę możemy znaleźć tu: `https://www.tcl.tk/man/tcl8.6/TkCmd/colors.htm`.

Pamiętaj, że nazwy kolorów podajemy w cudzysłowach i muszą się one dokładnie zgadzać z tymi na liście. Znaczenie mają także wielkie litery i spacje!

W ten sposób możemy łatwo rysować linie w kilku głównych kolorach, jednak nie będziemy w stanie wykorzystać pełnych możliwości ekranu komputera. Do tego potrzebna będzie inna metoda.

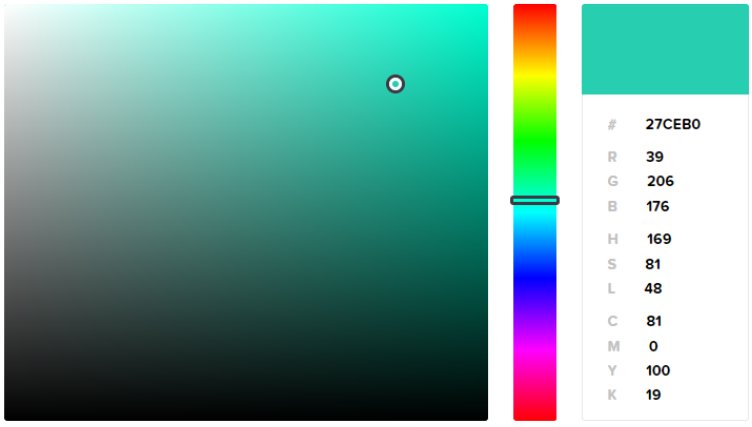
Zamiast podawać nazwę koloru, możemy podać intensywności barw czerwonej, zielonej i niebieskiej. Umożliwi nam to uzyskanie każdej z 16 milionów kombinacji które obsługuje standardowy kolowory wyświetlacz. Intensywności podajemy jako liczby pomiędzy 0 a 1, a im wyższa wartość, tym więcej danej barwy w mieszance. Dla przykładu, kolor czerwony możemy ustawić tak:

```
1 turtle.color(1.0, 0.0, 0.0);
```

Spróbuj!

Jakie wartości dadzą nam kolor zielony? Niebieski? Jakie barwy musimy zmieszać, aby otrzymać żółty, błękitny lub różowy? Jak otrzymać czerń, biel i szarości?

Oczywiście człowiekowi ciężko jest wyobrazić sobie jako kolor powstanie z danego połączenia podstawowych barw. Może się wydawać, że utrudnia to znacząco wybór odpowiednich wartości. Na szczęście istnieją narzędzia, które pozwalają na wybranie koloru z palety i odczytanie odpowiadających im liczb. Możemy użyć na przykład `http://htmlcolorcodes.com`. Tęczowy suwak pozwoli nam wybrać odpowiedni kolor, a paleta po lewej stronie jego odcień. W tabelce po prawo możemy odczytać wartości R, G i B, które odpowiadają kolejno czerwonemu, zielonemu i niebieskiemu.



Uwaga!

Podana wyżej strona, jak i większość dostępnych tego typu narzędzi, podaje wartości dla podstawowych barw jako liczby całkowite od 0 do 255. Aby otrzymać wartości między 0 a 1, których oczekuje żółw, wystarczy podzielić wszystkie trzy liczby przez 255.

```
turtle.color(39/255, 206/255, 176/255)
```

Alternatywnie, możemy zmienić tryb koloru żółwia tak, aby również przyjmował wartości od 0 do 255. Służy do tego polecenie `color_mode`.

```
turtle.colormode(255)
turtle.color(39, 206, 176)
```

Wypełnianie rysowanych kształtów

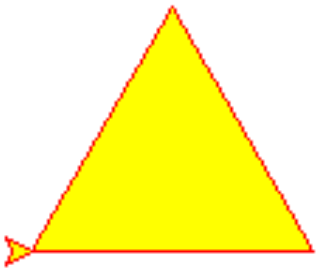
Oprócz zmiany koloru linii, możemy także rysowane kształty wypełniać kolorem. Aby to uczynić, musimy najpierw powiedzieć żółwiowi, że zaczynamy rysować kształt do wypełnienia, a po jego narysowaniu powiedzieć, że skończyliśmy. Służą do tego polecenia `begin_fill` i `end_fill`.

```
import turtle

turtle.begin_fill()
turtle.forward(100)
turtle.left(120)
turtle.forward(100)
turtle.left(120)
turtle.forward(100)
turtle.left(120)
turtle.end_fill()
```

Domyślnie kolor wypełnienia jest czarny, tak samo jak domyślny kolor linii. Polecenie `color` zmienia obie te wartości jednocześnie. Oczywiście możemy wybrać kolor wypełnienia niezależnie od koloru linii. Służą do tego odpowiednio polecenia `fillcolor` i `pencolor`.

```
import turtle

turtle.pencolor(1.0, 0.0, 0.0)
turtle.fillcolor(1.0, 1.0, 0.0)
```

```
5
6  turtle.begin_fill()
7  turtle.forward(100)
8  turtle.left(120)
9  turtle.forward(100)
10 turtle.left(120)
11 turtle.forward(100)
12 turtle.left(120)
13 turtle.end_fill()
```



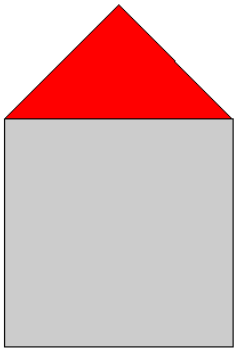
Podsumowanie

- Ludzkie oko reaguje na 3 kolory: czerwony, zielony i niebieski. Nazywamy je podstawowymi.
- Mózg jest w stanie odtworzyć dowolny kolor tęczy na podstawie różnic w intensywności między trzema podstawowymi kolorami.
- Komputery wykorzystują ten fakt aby wyświetlać niemal wszystkie kolory widoczne dla człowieka.
- Kolor linii rysowanej przez żółwia możemy zmieniać przy użyciu procedury `color`.
- Możemy podać nazwę koloru lub odpowiadające mu wartości 3 podstawowych barw.
- Możemy również wypełniać rysowane kształty kolorem używając procedur `begin_fill` i `end_fill`.
- Kolor wypełnienia i linii możemy zmieniać niezależnie procedurami `fillcolor` i `pencolor`.

Zadania

1. Stwórz program `rgb_squares.py` (lub `kwadraty_rgb.py`), który narysuje poniższy rysunek. Podpowiedź: stwórz odpowiednią funkcję, która będzie rysować kwadrat.

2. Stwórz program `colored_house.py` (lub `pokolorowany_dom.py`), który narysuje poniższy rysunek. Podpowiedź, krawędź dachu jest 1.41 razy dłuższa niż krawędź domku. Jeżeli chcesz, możesz dodać do niego dodatkowe elementy, jak drzwi czy okno.



3. Stwórz program `polish_flag.py` (lub `flaga_polski.py`), który narysuje flagę Polski. Według ustawy, wymiary flagi mają proporcje 5:8, kolor biały opisują wartości (233, 232, 231), a czerwony wartości (212, 33, 61).

4. Spróbuj narysować inne proste flagi, na przykład niemiecką, rumuńską, szwajcarską lub szwedzką. Ambitni mogą spróbować narysować flagę brytyjską.

6. Argumenty procedur

Procedury są niezwykle przydatnym narzędziem, jednak ich wadą jest to, że zawsze generują dokładnie ten sam rysunek. Kwadrat zawsze będzie miał ten sam bok, strzałka zawsze tę samą długość.

Dzisiaj nauczymy się jak do naszych procedur dodać parametry. Pozwolą one nam uzależnić wygląd tworzonego rysunku od pewnych początkowych wartości, na przykład wymaganej długości boku kwadratu, czy długości strzałki.

Uwaga!

Od teraz fragmenty kodu podawane jako przykłady rzadko będą pełnymi programami. Pomijane będą rzeczy które muszą być w każdym programie, jak `import turtle`. Pomijane będą definicje procedur, które były zdefiniowane wcześniej. Zakładam, że na tym etapie nauki będziesz już rozumieć jak podane fragmenty kodu się ze sobą łączą i jak stworzyć z nich pełen program.

Kwadrat

Zaczniemy od stworzenia prostej procedury rysującej kwadrat o boku 100.

```
1 def square():
2     turtle.forward(100)
3     turtle.left(90)
4     turtle.forward(100)
5     turtle.left(90)
6     turtle.forward(100)
7     turtle.left(90)
8     turtle.forward(100)
9     turtle.left(90)
```

Co jednak, gdybyśmy chcieli stworzyć procedurę rysującą kwadrat o boku 200? Musielibyśmy drugi raz skopiować ten sam kod, zmienić wszystkie długości boków na 200, a do tego nadać naszej procedurze nową nazwę.

```
1 def square_100():
2     turtle.forward(100)
3     turtle.left(90)
4     ...
5
6 def square_200():
7     turtle.forward(200)
8     turtle.left(90)
9     ...
```

Na szczęście istnieje prostsze rozwiązanie. Zamiast używać wewnątrz procedury konkretnych wartości, możemy użyć argumentu, którego wartość zostanie podana dopiero w momencie użycia naszej procedury.

Zwróć uwagę, że umiesz już *używać* procedur z parametrami. Niektóre z poleceń żółwia wymagają podania pewnych wartości, jak `forward` czy `left`.

Liczby podawane w nawiasach to właśnie argumenty. Teraz nauczymy się jak tworzyć takie procedury.

Chcielibyśmy móc rysować kwadraty w ten sposób:

```
square(100)
square(200)
```

W tym celu w definicji naszej procedury dodajemy w nawiasach nazwę dla naszego argumentu. Na przykład:

```
def square(side_length):
    ...
```

Nazwa argumentu może być niemal dowolna, jednak tak samo jak nazwa procedury, nie może zawierać spacji, nie powinna zawierać polskich liter, a do tego powinna opisywać to, do czego służy dany argument. Tutaj możemy nazwać argument `side_length`, po polsku "długość boku", lub po prostu `side` (bok).

Teraz możemy używać naszego argumentu wewnątrz kodu procedury, na przykład zamiast iść do przodu o 100 kroków, możemy iść do przodu o `side_length` kroków:

```
def square(side_length):
    turtle.forward(side_length)
    turtle.left(90)
    turtle.forward(side_length)
    turtle.left(90)
    turtle.forward(side_length)
    turtle.left(90)
    turtle.forward(side_length)
    turtle.left(90)
```

Spróbuj!

Stwórz nowy plik i dodaj do niego powyższą procedurę. Spróbuj narysować kilka kwadratów o różnych długościach boków.

O ile nazwa argumentu w nawiasach może być niemal dowolna, o tyle nazwa argumentu używana wewnątrz procedury musi zgadzać się z tą podaną na początku w nawiasach. Inaczej komputer nie zrozumie do którego argumentu się odnosimy.

Należy też pamiętać, że argument istnieje tylko wewnątrz procedury, którą definiujemy. Na przykład poniższy kod nie zadziała:

```
def square(side_length):
    turtle.forward(side_length)
    turtle.left(90)
    turtle.forward(side_length)
    turtle.left(90)
    turtle.forward(side_length)
    turtle.left(90)
    turtle.forward(side_length)
    turtle.left(90)

turtle.forward(side_length)
```

Jeżeli spróbujemy go uruchomić, zobaczymy następujący błąd:

```
NameError: name 'x' is not defined
```

Jeżeli się chwilę nad tym zastanowimy, to ma to sens. O jaką odległość powinien przemieścić się żołw w ostatniej linijce, gdyby ten kod był prawidłowy? Argument procedury zawiera wartość tylko przy wywołaniu tej procedury i nie może być używany poza nią.

Dzięki temu też nazwy argumentów mogą się powtarzać w różnych procedurach. Gdybyśmy stworzyli drugą procedurę, `triangle`, która rysuje trójkąt równoboczny, to ona również może zawierać argument o nazwie `side_length`.

```
def square(side_length):
    turtle.forward(side_length)
    turtle.left(90)
    ...

def triangle(side_length):
    turtle.forward(side_length)
    turtle.left(120)
    ...
```

Wiele argumentów

A co gdybyśmy chcieli stworzyć procedurę rysującą prostokąty? Prostokąt posiada boki o dwóch różnych długościach, jak możemy sobie z tym poradzić?

Otóż procedury mogą posiadać kilka argumentów. Oddzielamy je wtedy przecinkami.

```
def rectangle(width, height):
    turtle.forward(width)
    turtle.left(90)
    turtle.forward(height)
    turtle.left(90)
    turtle.forward(width)
    turtle.left(90)
    turtle.forward(height)
    turtle.left(90)
```

Kolejność argumentów ma znaczenie. W momencie użycia takiej procedury, wartości zostaną przypisane argumentom w takiej kolejności, w jakiej zostały podane. Dlatego przy wywołaniu:

```
rectangle(100, 200)
```

argument `width` otrzyma wartość 100, a `height` 200.

Nazwy argumentów w jednej procedurze nie mogą się powtarzać. Co oczywiście też ma sens, gdybyśmy dwa argumenty nazwali tak samo, skąd komputer miałby wiedzieć do którego z nich się odnosimy?

```
def rectangle(side, side):
    turtle.forward(side)
    ...
```

Próba uruchomienia powyższego kodu da nam poniższy błąd:

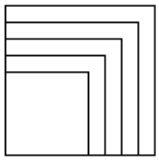
```
SyntaxError: duplicate argument 'side'
    in function definition
```

Podsumowanie

- Procedury mogą przyjmować argumenty, czyli wartości które zostaną podane dopiero w momencie wywołania procedury.
- Używaliśmy już tego typu procedur wcześniej, na przykład `forward` czy `left`.
- Aby stworzyć taką procedurę, w nawiasach po jej nazwie podajemy nazwy argumentów.
- Aby użyć wartości argumentu podanej przy wywołaniu, wpisujemy po prostu nazwę tego argumentu w kodzie.
- Argument istnieje tylko wewnątrz procedury, która go zdefiniowała. Dzięki temu możemy w kilku procedurach użyć tej samej nazwy argumentu.
- Jeżeli procedura posiada kilka argumentów, to ich nazwy muszą być unikalne, a ich kolejność ma znaczenie.

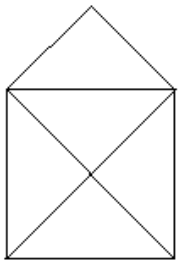
Zadania

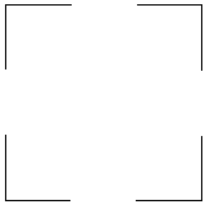
1. Używając procedury `square(side_length)` zdefiniowanej w tym rozdziale, stwórz program, który narysuje poniższy rysunek.



2. Stwórz procedurę `arrow(length)`, która narysuje prostą strzałkę o długości `length`. Rozmiar grota może być dowolny. Co stanie się, jeżeli spróbujesz narysować strzałkę, która jest krótsza niż grot?

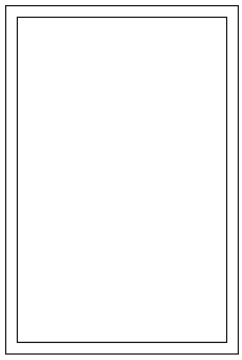
3. Stwórz procedurę `envelope(size)`, która narysuje otwartą kopertę jak na rysunku poniżej. Argument `size` opisuje długość boku koperty. Podpowiedź: przekątna koperty jest 1.41 razy dłuższa niż bok koperty.
4. Stwórz procedurę `broken_square(side, gap)`, która narysuje przerwany kwadrat jak na rysunku poniżej. Argument `side` opisuje długość boku kwadratu, a `gap` długość przerwy. Przerwa powinna znajdować się na środku boku. Przykład przedstawia wynik wywołania `broken_square(100, 33)`.





5. Stwórz procedurę `frame(width, height, gap)`, która narysuje ramkę taką jak poniżej. Argument `width` opisuje szerokość ramki, `height` jej wysokość, a `gap` odległość między zewnętrznym a wewnętrznym prostokątem. Przykład poniżej powstał po wywołaniu `frame(100, 200, 10)`.

   Podpowiedź: stwórz najpierw procedurę `rectangle(width, height)`, która narysuje prostokąt.



7. Powtarzanie

Dzięki procedurom i funkcjom możemy wykorzystać ten sam fragment kodu w kilku miejscach w programie. Czasem jednak chcemy ten sam kod wykonać kilka razy pod rząd w jednym miejscu. Możemy co prawda stworzyć odpowiednią procedurę i skopiować jej wywołanie kilkukrotnie, jednak może to utrudniać zrozumienie kodu.

W tym rozdziale nauczymy się używać pętli, które pozwolą nam powtórzyć wybrany fragment kodu pewną znaną z góry liczbę razy. Ułatwi to tworzenie rysunków takich jak wielokąty czy gwiazdy.

Pętla `for`

Python, jak niemal każdy język programowania, posiada pętle. Istnieje kilka różnych rodzajów pętli, każdy z nich przydatny w innych sytuacjach. Dzisiaj poznamy pętlę `for`, która pozwoli nam potwórzyć wybrany fragment kodu daną liczbę razy.

Wygląd tej struktury może wydawać się nieco dziwny, jednak dlaczego wygląda ona tak a nie inaczej dowiemy się później.

Jako przykład znów posłuży nam kwadrat.

```
1 for _ in range(4):
2     turtle.forward(100)
3     turtle.left(90)
```

Liczba w nawiasach po `range` mówi nam ile razy dany kod zostanie powtórzony. Podobnie jak w przypadku procedur, pierwsza linijka kończy się dwukropkiem, a kolejne linijki są wcięte. Dzięki wcięciom możemy dokładnie powiedzieć komputerowi gdzie kończy się fragment kodu, który chcemy powtórzyć.

```
1 turtle.color("red")
2 turtle.begin_fill()
3 for _ in range(4):
4     turtle.forward(100)
5     turtle.left(90)
6 turtle.end_fill()
```

Oczywiście pętle możemy również umieszczać w procedurach, na przykład:

```
1 def square(side):
2     for _ in range(4):
3         turtle.forward(side)
4         turtle.left(90)
```

Zwróć uwagę na to, że kod wewnątrz pętli jest teraz wcięty o dwa poziomy!

Do czego to podkreślenie?

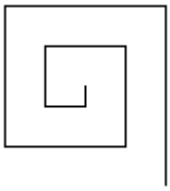
Pętle `for`, podobnie jak procedury, wykonywane są z pewnym argumentem. Dotychczas w miejsce nazwy argumentu umieszczaliśmy znak podkreśle-

nia, jako informację o tym, że argument ten nie będzie wykorzystywany. W zamian możemy tam umieścić inną nazwę, na przykład `side`.

Jaka jednak będzie wartość tego argumentu? Przyjrzyjmy się jak wygląda i co oznacza pierwsza linjka poniższej pętli.

```
for side in range(10):
    turtle.forward(side * 10)
    turtle.right(90)
```

`for side in range(10)` jest pewnego rodzaju skrótem myślowym od "for each value side in range from 0 do 10", co po polsku oznacza "dla każdej wartości `side` w zakresie od 0 do 10". Oznacza to, że każde kolejne wykonanie pętli zostanie wykonane z kolejną liczbą naturalną od 0 do 10. Powyższa pętla narysuje więc taką spiralę:



Jeżeli jednak uważnie przyjrzymy się naszej spirali, to zobaczymy że składa się nie z 10 odcinków, a z 9. Jest tak dlatego, że za pierwszym razem pętla wykona się z argumentem 0, co narysuje odcinek zerowej długości i obróci żółwia w prawo. Z tego samego powodu najkrótszy odcinek spirali jest pionowy, a nie poziomy.

Aby rozwiązać ten problem, możemy do wartości `side` dodać 1. Jest jednak lepszy, bardziej ogólny sposób na rozwiązanie tego problemu.

Zakresy

Używana tutaj funkcja `range(n)` tworzy zakres liczb od 0 do `n`, z wyłączeniem `n`. Stąd `range(3)` da nam 0, 1, 2.

Spróbuj!

Działanie funkcji `range` najlepiej zrozumieć na przykładach. Możemy zobaczyć jaki zakres powstanie w wyniku jakiego wywołania przy użyciu konsoli Pythona. Jeżeli ją otworzysz i wpiszesz `range(10)`, wynik nie będzie zbyt przydatny...

```
>>> range(10)
range(10)
```

Możemy jednak wylistować elementy zakresu, przekazując go do innej funkcji, `list`:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Istnieje jednak kilka innych wersji funkcji `range`, które będą tworzyć inne zakresy. Jeżeli zamiast jednej liczby podamy dwie, to pierwsza będzie

wartością startową, a druga końcową, tak jak w przypadku podania jednego argumentu.

```
>>> list(range(2, 5))
[2, 3, 4]
```

W końcu możemy podać 3 argumenty. Trzeci argument będzie oznaczać wtedy krok, czyli różnicę między kolejnymi elementami zakresu.

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

Możemy wykorzystać tę wartość na dwa sposoby. Po pierwsze, możemy zwiększyć krok, co pozwoli nam uniknąć mnożenia. Na przykład spiralę z poprzedniej sekcji możemy zapisać tak:

```
for side in range(0, 100, 10):
    turtle.forward(side)
    turtle.right(90)
```

Spróbuj!

Co stanie się, jeżeli krok nie będzie dzielnikiem różnicy między początkiem a końcem? Jakie wartości zwróci na przykład `range(0, 10, 4)`?

Dzięki zmianie kroku możemy też odwrócić kolejność zakresu. Jeżeli krok będzie ujemny, to kolejne wartości będą coraz mniejsze.

```
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
```

W ten sposób możemy na przykład narysować spiralę od zewnątrz:

```
for side in range(100, 0, -10):
    turtle.forward(side)
    turtle.right(90)
```

Uwaga!

Zwróć uwagę, że dla ujemnego kroku, początek zakresu musi być większy niż koniec. W przeciwnym wypadku pętla nie wykona się ani razu. Funkcja `range` zwróci pusty zakres, bo nie jest w stanie otrzymać większej liczby odejmując wartości od mniejszej.

Funkcja `range` nie działa jednak na ułamkach. Często zdarza się jednak, że chcemy aby przynajmniej wielkość kroku była ułamkiem (na przykład w wyniku dzielenia dwóch wartości). W takiej sytuacji musimy z wyniku działania zrobić liczbę całkowitą, używając funkcji `int`.

```
for size in range(height, 0, int(height/levels)):
    ...
```

Listy

Jest jeszcze jeden sposób na wybranie argumentów dla kolejnych wywołań pętli. Zamiast używać funkcji `range`, możemy samodzielnie wybrać kolejne wartości i podać je w formie listy.

Więcej informacji...

Wybieranie wartości dla pętli nie jest oczywiście jedynym zastosowaniem list, jednak o tym do czego jeszcze się przydają dowiemy się nieco później.

Lista to uporządkowany zestaw wartości. Oznacza to, że wartości jest wiele, a ich kolejność ma znaczenie. Listy zapisujemy używając nawiasów kwadratowych, a w środku podajemy wartości oddzielone przecinkami. Tak jak to widzieliśmy w poprzedniej sekcji używając funkcji `list`, aby wypisać wartości zakresu. Funkcja ta tworzyła właśnie listę.

Dzięki listom możemy ułatwić sobie na przykład rysowanie prostokątów:

```
1 for side in [100, 200, 100, 200]:
2     turtle.forward(side)
3     turtle.right(90)
```

Ale listy pozwalają na dużo więcej. Nie muszą na przykład wcale zawierać liczb, a na przykład kolory!

```
1 for color in ["red", "green", "blue"]:
2     turtle.color(color)
3     turtle.begin_fill()
4     square(100)
5     turtle.end_fill()
6     turtle.forward(100)
```

Więcej informacji...

Jeżeli chcesz umieścić na liście kolory w postaci RGB, każdy kolor umieść w okrągłych nawiasach, na przykład:

```
1 for color in [(1, 1, 0), (1, 0, 1), (0, 1, 1)]:
2     turtle.color(color)
3     turtle.begin_fill()
4     square(100)
5     turtle.end_fill()
6     turtle.forward(100)
```

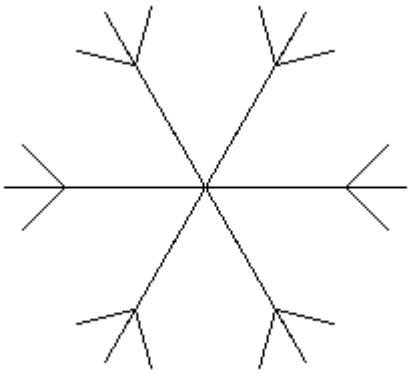
Podsumowanie

- Możemy powtarzać fragment kodu kilka razy używając pętli `for`.
- Kolejne wykonania pętli nie muszą być identyczne; pętle również przyjmują argumenty.

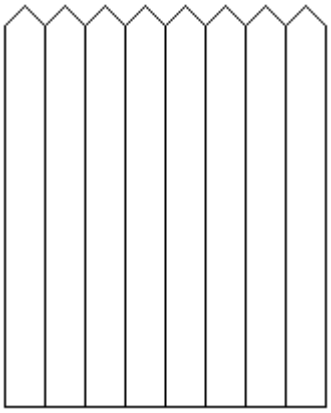
- Argumenty mogą być wybrane z zakresu stworzonego funkcją `range`.
- Mogą też być dowolną listą wartości, nawet kolorów.

Zadania

1. Napisz procedurę `triangle(side)`, która narysuje trójkąt równoboczny o boku `side`.
2. Napisz program, który narysuje okrąg. Rozmiar okręgu jest dowolny.
3. Napisz program, który narysuje płatek śniegu jak poniżej.

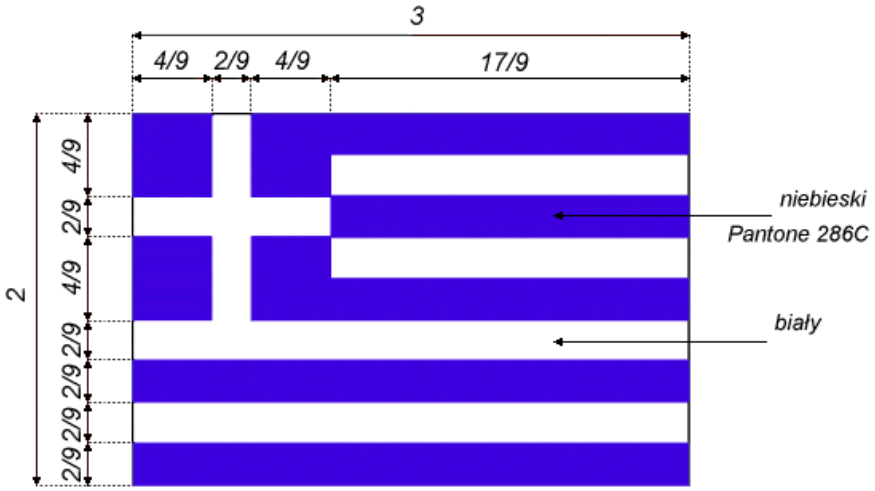


4. Napisz program, który narysuje płot jak poniżej. Liczba i wymiar sztachet są dowolne.



5. Napisz program, który narysuje flagę Grecji. Poniżej względne wymiary flagi i jej elementów.
6. Napisz procedurę `polygon(n, side)`, która narysuje wielokąt foremny o liczbie boków `n` i długości boku `side`. Kąt wewnętrzny wielokąta foremnego o n bokach można policzyć według poniższego wzoru

$$\frac{(n-2)\cdot 180°}{n}$$



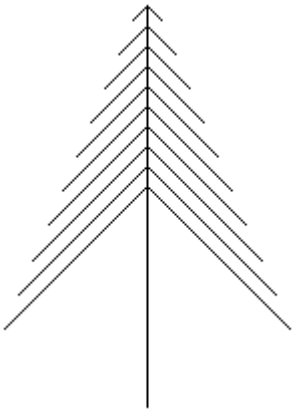
Źródło: Wikimedia Commons

7. Napisz procedurę `circle(size)`, która narysuje okrąg o średnicy `size`. Podpowiedź: zdecydowanie łatwiej jest narysować okrąg o zadanym obwodzie. Aby otrzymać długość obwodu, przemnóż średnicę przez 3.14.

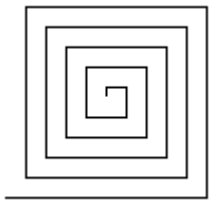
8. Napisz procedurę `fence(n, width, height)`, która narysuje płot jak w zadaniu 4, który będzie miał `n` sztachet, wysokość `height`, i szerokość `width`.

   Podpowiedź: zacznij od procedury `plank(width, height)`, która narysuje jedną sztachetę o zadanych wymiarach.

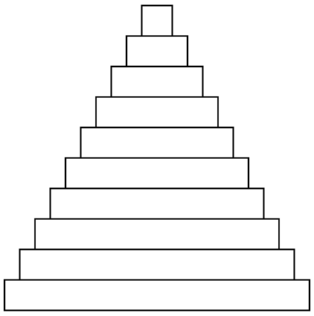
9. Napisz procedurę `tree(n)`, która narysuje choinkę jak poniżej, posiadającą `n` gałęzi. Poniżej wynik działania `tree(10)`.



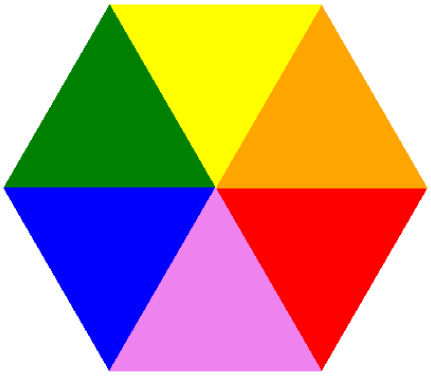
10. Napisz procedurę `square_spiral(n, size)`, która narysuje kwadratową spiralę jak na rysunku poniżej. Spirala powinna składać się z `n` odcinków. Najdłuższy odcinek powinien mieć długość `size`. Poniżej przykład wywołania `square_spiral(10, 200)`.



11. Napisz procedurę `pyramid(levels, height)`, która narysuje piramidę jak na rysunku poniżej. Argument `levels` oznacza liczbę poziomów piramidy, a `height` jej wysokość, która jest równa także szerokości podstawy piramidy. Poniżej wynik działania `pyramid(10, 200)`.



12. Napisz program, który narysuje kolorowy sześciokąt jak poniżej.



8. Moduły

Na pewno zdążyliście już zauważyć, że wiele procedur tworzonych na potrzeby jednego rysunku przydaje się potem w innych. Oczywiście, możemy je za każdym razem pisać od nowa albo przekopiować z wcześniejszego projektu. Istnieje jednak dużo lepszy sposób, moduły.

Moduł będzie zawierać przydatne procedury, które będziemy mogli potem zaimportować do naszych programów. W tym rozdziale nauczymy się tworzyć i używać własne moduły. Przy okazji dowiemy się co dokładnie do czego służy `import turtle`!

Tworzenie modułu

Tak naprawdę to już wiecie jak tworzyć moduły. W Pythonie moduł to nic innego jak kolejny plik `.py`. Najlepiej, żeby nie posiadał żadnego kodu, który zostanie wykonany przy jego uruchomieniu, jedynie definicje procedur. W przeciwnym wypadku, cały kod zostanie wykonany w momencie importowania pliku.

Aby zobaczyć jak to wygląda w praktyce, stworzymy moduł `geometry` (lub `geometria`) i umieścimy w nim procedury rysujące kilka prostych kształtów.

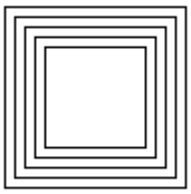
Zaczniemy od stworzenia pliku `geometry.py`, w którym umieścimy procedurę rysującą kwadrat:

```
1 import turtle
2
3 def square(side_length):
4     for _ in range(4):
5         turtle.forward(side_length)
6         turtle.left(90)
```

...i w ten sposób stworzyliśmy nasz pierwszy moduł!

Używanie własnego modułu

Teraz wykorzystamy nasz moduł do narysowania poniższego rysunku:



Stworzymy nowy plik, `squares.py` i w środku umieścimy kod, który narysuje powyższy rysunek. Aby użyć naszego nowo stworzonego modułu w programie, musimy go najpierw zaimportować. Tak jak wcześniej robiliśmy to z modułem `turtle`, tak teraz to samo musimy zrobić z modułem `geometry`.

Uwaga!

Jako że nadal chcemy używać poleceń żółwia, to musimy zaimportować oba moduły!

```
import geometry
import turtle

...
```

Teraz wszystkie procedury zdefiniowane w pliku `geometry.py` są dostępne w naszym programie. Jednak podobnie jak robiliśmy to z poleceniami żółwia, przed każdym wywołaniem procedury z modułu `geometry` musimy umieścić nazwę modułu i kropkę.

```
import geometry
import turtle

for side_length in range(50, 100, 10):
    turtle.down()
    geometry.square(side_length)
    turtle.up()
    turtle.backward(5)
    turtle.right(90)
    turtle.forward(5)
    turtle.left(90)
```

Tworzenie modułu tylko po to, żeby użyć go w jednym programie może wydawać się zbędnym wysiłkiem, przecież łatwiej byłoby po prostu umieścić tę jedną procedurę w naszym programie. Jednak następnym razem, kiedy będziemy potrzebować procedury rysującej kwadrat, nie będziemy musieli już jej pisać od nowa ani kopiować. Wystarczy wpisać na górze programu `import geometry`.

Poza tym moduły pozwalają na rozbicie większych programów na mniejsze, a co za tym idzie łatwiejsze do zrozumienia, części. Umożliwia to także testowanie części naszego programu bez uruchamiania całości, co ułatwia znajdowanie błędów w większych programach. O tym jednak więcej dowiemy się nieco później.

Podsumowanie

- Moduł to plik z kodem Pythona, który zawiera, między innymi, definicje przydanych procedur.
- Tworzenie modułu nie różni się niczym od tworzenia nowego pliku z kodem.
- Aby użyć modułu w programie, należy go zaimportować.
- Dzięki modułom możemy uniknąć pisania kilka razy tych samych procedur.
- Rozbijanie dużych programów na moduły ułatwia ich testowanie.

Zadania

Rozszerz moduł geometry o inne przydatne procedury, na przykład:

1. `rectangle(width, height)`, rysującą prostokąt o zadanej szerokości i wysokości
2. `polygon(n, side)`, rysujący wielokąt foremny o `n` bokach, każdy o długości `side`
3. `star(size)`, rysujący pięcioramienną gwiazdę o ramieniu długości `side`.

Przejrzyj wszystkie swoje wcześniej napisane programy i znajdź miejsca w których można użyć powyższych procedur. Zastąp zdefiniowane w nich procedury wywołaniami do odpowiednich procedur w module `geometry`. Może przy okazji znajdziesz inne procedury, które często się powtarzają i pasowałyby do tego modułu? A może lepiej pasowałyby do modułu o innej nazwie? Stwórz go!

9. Szybsze rysowanie

Być może udało ci się zauważyć, że rysowanie bardziej skomplikowanych kształtów może zająć żółwiowi sporo czasu. Oczywiście, czasem może być to przydatne, szczególnie kiedy chcemy znaleźć błąd w programie. Jednak czekanie po kilkanaście czy nawet kilkadziesiąt sekund za każdym razem kiedy uruchomimy program może być frustrujące.

W tym rozdziale dowiemy się nieco o tym jak działa animacja, jaki związek ma to z szybkością żółwia, a na koniec poznamy kilka sztuczek, które pozwolą nam znacząco przyspieszyć uruchamiane programy.

Komputery a filmy

Z jednego z poprzednich rozdziałów wiemy już nieco o tym jak komputery przechowują obrazy. Dla przypomnienia, obraz jest dla komputera siatką wielu pikseli, czyli małych punktów, z których każdy ma dokładnie jeden kolor. W nowoczesnych urządzeniach piksele te są tak małe, że ciężko jest je dostrzec nawet jeżeli ekran jest kilka centrymetrów od naszych oczu.

Film natomiast składa się z ciągu obrazów, które wyświetlane są po kolei, w bardzo szybkim tempie. Już 25 obrazów (lub klatek) na sekundę jest w stanie oszukać ludzkie oko i stworzyć złudzenie ruchu. Może się on jednak wydawać mało płynny – najlepsze efekty otrzymuje się przy 60 i więcej klatkach na sekundę.

Oznacza to, że komputer ma około 17 milisekund na obliczenie koloru każdego piksela na ekranie. Z jednej strony to mało – przeciętne mrugnięcie trwa od 300 do 400 milisekund. Z drugiej jednak strony nowoczesnemu komputerowi wystarczy to na wykonanie kilku miliardów operacji. Trzeba jednak pamiętać, że obraz może składać się z kilku millionów pikseli (obraz w formacie 4K posiada ich aż 8 milionów), i kolor każdego z nich trzeba osobno policzyć.

Jeżeli komputer nie jest w stanie obliczać nowych klatek wystarczająco szybko, liczba klatek na sekundę zacznie spadać. To dlatego bardziej wymagające gry komputerowe mogą nie działać zbyt płynnie na starszych komputerach – ilość obliczeń do wyświetlenia całej klatki jest zbyt duża, aby komputer zdążył je wszystkie wykonać na czas.

Jak ma się to do żółwia

Domyślnie, żółw wykonuje jedynie jedno polecenie na klatkę, a do tego klatki wyświetlane są nie częściej niż co 10 milisekund. Oznacza to, że wykonanie 100 poleceń zajmie co najmniej sekundę.

Spróbujmy zatem oszacować ile zajmie narysowanie okręgu w następujący sposób:

```
1 for _ in range(360):
2     turtle.forward(1)
3     turtle.left(1)
```

Pętla wykona się 360 razy i za każdym razem wykona dwa polecenia, co daje w sumie 720 poleceń. Przy 10 milisekundach na polecenie, dostajemy 7.2 sekundy na cały rysunek.

W praktyce oczywiście może zająć to więcej czasu. Spróbujmy zmierzyć to w praktyce!

Mierzenie czasu

Python posiada wbudowane procedury umożliwiające mierzenie czasu. Znajdują się one w module `time`, musimy więc go zaimportować.

Interesuje nas głównie procedura `perf_counter`, która zwraca czas w sekundach, z bardzo dużą dokładnością, dlatego bardzo często nie będzie to liczba całkowita.

Aby zmierzyć czas potrzebny na wykonanie fragmentu kodu, musimy pobrać obecny czas tuż przed wykonaniem tego kodu i tuż po, a następnie policzyć różnicę.

```
1 import time
2 import turtle
3
4 start = time.perf_counter()
5 for _ in range(360):
6     turtle.forward(1)
7     turtle.left(1)
8 end = time.perf_counter()
9 print(end - start)
```

Przed narysowaniem koła zapamiętujemy obecny czas zapisując go w zmiennej. Nieco więcej o zmiennych dowiemy się w późniejszym rozdziale. Podobnie robimy na końcu rysowania koła, aby zapisać czas końcowy. Na końcu wypisujemy różnicę tych dwóch wartości, co daje nam całkowity czas potrzebny na wykonanie rysunku.

Kiedy uruchomimy ten program, najpierw zobaczymy jak żółw rysuje okrąg, a następnie okno zamknie się. Na dole ekranu PyCharm Edu zobaczymy jednak czas, który potrzebny był na narysowanie naszego rysunku. W moim przypadku było to 13.5s.

Przyspieszamy żółwia

Jak jednak wykorzystać tę wiedzę do przyspieszenia rysunków żółwia? Możemy to zrobić na dwa sposoby. Po pierwsze, możemy zmniejszyć opóźnienie pomiędzy kolejnymi klatkami. Służy do tego procedura `delay`, która przyjmuje jeden argument: mininmalny czas w milisekundach, który powinien upłynąć pomiędzy kolejnym klatkami. Domyślnie jest to 10 – możemy przekonać się o tym ustawiając opóźnienie na 10, zanim zaczniemy rysować okrąg.

```
1  import time
2  import turtle
3
4  turtle.delay(10)
5
6  start = time.perf_counter()
7  for _ in range(360):
8      turtle.forward(1)
9      turtle.left(1)
10 end = time.perf_counter()
```

```
print(end - start)
```

Pomiar powinien być podobny jak w pierwszym przypadku. Teraz możemy to opóźnienie zmniejszyć, na przykład o połowę. Powinno to skrócić czas potrzebny na wykonanie rysunku o około połowę. Tak było w moim przypadku: czas wykonania skrócił się z 13.5 sekundy do 7 sekund.

Możemy oczywiście pójść o krok dalej i ustawić opóźnienie na 0. W moim przypadku wykonanie rysunku zajęło wtedy około pół sekundy.

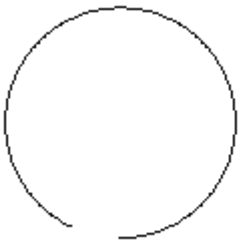
Drugi sposób, to zmuszenie komputera, aby nie wykonywał tylko jednego polecenia żółwia na sekundę. Służy do tego polecenie `tracer`, którego argumentem jest liczba poleceń która zostanie wykonana przed każdym odświeżeniem ekranu. Domyślnie będzie to 1, a żeby przyspieszyć rysowanie, możemy tę wartość zwiększyć.

```
import time
import turtle

turtle.delay(0)
turtle.tracer(2)

start = time.perf_counter()
for _ in range(360):
    turtle.forward(1)
    turtle.left(1)
end = time.perf_counter()
print(end - start)
```

Na moim komputerze zmniejszyło to czas potrzebny na wykonanie rysunku około trzykrotnie. Możemy oczywiście zwiększać tę wartość dowolnie, jednak musimy mieć jedną rzecz na uwadze. Jeżeli ekran będzie odświeżany zbyt rzadko, to część rysunku może być wykonana już po ostatnim odświeżeniu ekranu, a więc nie pokaże się w oknie.

Aby to zaobserwować, wywołajmy polecenie `tracer` z wartością 1000. Przyda się też dodanie na końcu programu `exitonclick` (po odczytaniu końcowego czasu!), ponieważ rysunek może wykonać się za szybko, żebyśmy w ogóle go zauważyli.

Na moim komputerze wykonanie rysunku zajęło niewiele więcej niż jedną setną sekundy, jednak okrąg nie był pełny:



Aby naprawić ten problem, możemy na końcu rysowania wymusić odświeżenie ekranu używając polecenia `update`.

```
import time
import turtle

turtle.delay(0)
```

```
turtle.tracer(1000)

start = time.perf_counter()
for _ in range(360):
    turtle.forward(1)
    turtle.left(1)
turtle.update()
end = time.perf_counter()
print(end - start)
```

Narysowanie okręgu nadal jest zbyt szybkie, żebyśmy je zauważyli, ale tym razek okrąg jest już cały.

Podsumowanie

- Komputery wyświetlają filmy i animacje poprzez wyświetlanie po kolei wielu obrazów, kilkadziesiąt na sekundę.
- Już 25 obrazów (klatek) na sekundę wystarczy, aby oszukać ludzkie oko, że widzi płynny ruch.
- Obliczenie wszystkich kolorów pikseli może zająć więcej niż kilkanaście milisekund, dlatego filmy w 4K lub bardziej wymagające gry mogą się zacinać.
- Żółw rysuje bardziej skomplikowane rysunki wolno, bo wykonuje co najwyżej jedno polecenie na klatkę, a do tego nie wyświetla więcej niż jednej klatki co 10 milisekund.
- Możemy zmienić te wartości używając procedur `delay` i `tracer`.
- Jeżeli ekran będzie odświeżać się zbyt rzadko, część rysunku może nie wyświetlić się na ekranie. Możemy temu zaradzić wymuszając odświeżenie ekranu procedurą `update`.

Zadania

Czy któryś z twoich rysunków wykonywał się nieznośnie wolno? Spróbuj zmierzyć czas, który był potrzebny na jego wykonanie. Spróbuj przyspieszyć jego rysowanie używając wiedzy z tego rozdziału.

10. Pętle zagnieżdżone

11. Instrukcje warunkowe

12. Rekurencja

13. Fraktale

14. Zmienne i stałe