PYTHONOWE WYZWANIA DLA POCZĄTKUJĄCYCH

Wanda Jochemczyk, Katarzyna Olędzka
Ośrodek Edukacji Informatycznej i Zastosowań Komputerów
02-026 Warszawa, Raszyńska 8/10
{wanda.jochemczyk, katarzyna.oledzka}@oeiizk.waw.pl

Abstract. Using the experience gained during teacher training we will present our proposal for teaching and learning programming in Python from the beginning. We will present some examples of the tasks that can be solved by students in the class. If we motivate students enough, they are ready to take the effort and take on the challenge.

1. Wstęp

Naszą znajomość z językiem Python rozpoczęłyśmy od poszukiwania darmowego środowiska programistycznego w którym uczniowie mogliby uczyć się programowania wykorzystując ideę grafiki żółwia. Dostępne implementacje Logo są albo płatne, albo dość proste i mało atrakcyjne dla dzieci. Grafika żółwia jest jednym z pomysłów na początki programowania, stąd potrzeba znalezienia odpowiedniego środowiska, w którym można sterować obiektem po ekranie. Python z modułem `turtle` dostarcza potrzebnych narzędzi. Warto tutaj wspomnieć o licznym udziale uczniów w konkursach programistycznych Logia i miniLOGIA organizowanych na terenie województwa mazowieckiego. Uczniowie piszą programy m.in. w języku Python. Nie o tym jednak będziemy pisać.

Mając już za sobą doświadczenie, że młodzi ludzie chcą i mogą nauczyć się programować, szukamy nowych sposobów jak im pomóc. Dlatego w Ośrodku Edukacji i Zastosowań Komputerów w Warszawie przygotowaliśmy szkolenia, które pomogą w nauce programowania. Jednym z pomysłów jest nauka grafiki żółwia, innym – niejako niezależnym – rozwiązywanie problemów z wykorzystaniem operacji matematycznych oraz przetwarzania słów i list. Stąd powstały szkolenia dla nauczycieli: Programowanie w języku *Python – grafika żółwia*, *Programowanie w języku Python – algorytmika* oraz dla uczniów *Pythonowa drabina*. Naukę programowania opieramy na rozwiązywaniu zadań algorytmicznych o wzrastającym stopniu trudności.

2. Podejście do nauki programowania

Na początku warto stwierdzić, jakiego podejścia nie stosujemy. Po pierwsze, nie został przygotowany kompletny kurs języka Python, w którym uczący się poznaje wszystkie tajniki składni języka. Python, podobnie jak inne języki wysokiego poziomu, ma wiele konstrukcji programistycznych typowych – instrukcję warunkową, pętle, zmienne, … oraz specyficzne, charakterystyczne dla tego języka, niespotykane lub rzadko spotykane w innych językach. Na przykład równoległe przypisanie (`a, b = b%a, a`), iteratory i generatory (`yield`), czy też tzw. listy składane (`[len(slowo) for slowo in slowa]`). W zamian zostały przygotowane krótkie materiały, w których prezentowana jest wiedza potrzebna do rozwiązania różnych problemów. Materiały odpowiadają jednemu zagadnieniu, np. stosowanie zmiennych i są opracowane tak, by z jednej strony wprowadzić niezbędne konstrukcje językowe, z drugiej, by podać konkretne zastosowania. Całość opatrzona jest komentarzem, który pozwala ukierunkować myślenie na poznanie metod i narzędzi informatycznych, który są pomocne w rozwiązywaniu zadań i problemów.

Po drugie, nie preferujemy prezentacji długich i trudnych do zrozumienia algorytmów, które trzeba jedynie przepisać i starać się zrozumieć. Umiejętność przepisywania, znana już w starożytności, wymaga skupienia, dokładności i sumienności. Nie to jest jednak głównym przedmiotem zajęć informatycznych. Kształtując postawy twórcze, trzeba tak dobierać zadania, by pobudzać do myślenia. Zaczynamy od prostych problemów, by przejść do coraz trudniejszych. Znane algorytmy, które stanowią fundament algorytmiki, wprowadzamy ukazując ich zasadę działania, ale też i znaczenie oraz zastosowania. Staramy się tak wprowadzać modyfikacje zadań i algorytmów, by sprawdzić ich rozumienie i kształtować umiejętność rozwiązywania problemów z ich wykorzystaniem.

Po trzecie, unikamy podejścia, w którym liczy się tylko sukces i bycie najlepszym. Informatyka jest specjalnością młodych Polaków, ale nie tylko tych, którzy zdobywają laury na krajowych i międzynarodowych olimpiadach. Wielu osób o solidnej wiedzy matematycznej potrafi programować. Opracowują oni narzędzia wspomagające pracę ludzi i sterujące działaniem maszyn. Aby je przygotować potrzeba wszechstronnej wiedzy i niemałego wysiłku. Część z nich zaczęła swoją przygodę informatyczną, gdyż spotkała na swojej drodze kogoś, kto pokazał im swoją pasję.

Ogólnie, chociaż mamy do czynienia z zagadnieniami trudnymi, wymagającymi wielu umiejętności, a także dużej wiedzy matematycznej, możemy je przedstawić w ciekawej i przystępnej formie. W szkole pracujemy z uczniami, którzy są ciekawi świata. Jeśli ich dostatecznie zmotywujemy, są gotowi podjąć wysiłek. Przyjrzyjmy się kilku przykładowym zagadnieniom, które będziemy je rozwiązywać krok po

kroku. Zanim rozwiążemy postawiony problem, będziemy szukać rozwiązania dla zagadnień prostych, które pomogą w znalezieniu rozwiązania całego problemu. Zaprezentujemy kolejno trzy wyzwania: (1) zadanie o żuczku, który wchodzi na słup, (2) zadanie o liczeniu dni w roku oraz (3) zadanie o rysowaniu strzałki.

3. Wyzwanie 1 – żuczek, który wchodzi na słup

Zadanie

Napisz funkcję `kiedy(x,y)`, której wynikiem będzie liczba określająca dzień, kiedy mały żuczek znajdzie się na szczycie dziesięciometrowego słupa. Żuczek w dzień wspina się o x centymetrów, w nocy spada o y centymetrów. Załóż, że `x > y`.

Przykłady: Wynikiem `kiedy(300,100)` jest `5`. Wynikiem `kiedy(4,2)` jest `499`. Zadanie pochodzi z konkursu LOGIA 13 (etap 1).

Potrzebna wiedza

Uczeń powinien znać podstawowe operacje matematyczne: `+`, `-`, `*`, `/`, `//` (dzielenie całkowite), `%` (reszta z dzielenia) oraz wiedzieć jak definiować funkcje.

```
def nazwa(parametry):
    instrukcje
    return wyrażenie
```

Rozwiązanie

Zadanie będziemy rozwiązywać metodą kolejnych utrudnień. Najpierw założymy, że żuczek tylko się wspina i nie spada w nocy (czyli `y = 0`) oraz, że `1000` jest wielokrotnością `x`. Potem rozpatrzymy przypadki, w których niekoniecznie `1000` jest wielokrotnością `x`. Następnie rozwiążemy zadanie dla dowolnych `y` – najpierw gdy `1000` jest wielokrotnością `x-y`, potem bez tego ograniczenia, czyli pełne zadanie.

Zastanówmy się nad pytaniem, kiedy żuczek znajdzie się na szczycie dziesięciometrowego słupa, jeśli codziennie wspina się o `x` cm, bez spadania w nocy.

Przy założeniu, że `1000` jest wielokrotnością `x`, będzie to:

```
def kiedy(x):
    return 1000//x
```

Jeśli nie wiemy, czy `1000` jest wielokrotnością `x`, to zagadnienie jest trochę trudniejsze. Zakładamy, że nie korzystamy z innych funkcji matematycznych (np. tzw. `sufit` – czyli zaokrąglanie w górę). Pytanie pomocnicze, jakie możemy

przedstawić, to kiedy znajdzie się na wysokości, która pozwala w jeden dzień wejść na szczyt. Potem trzeba dodać jeden dzień, żeby żuczek wszedł na słup.

```
def kiedy(x):
    return (1000-1)//x+1
```

Nie trudno zauważyć, że jeśli w dzień wchodzi o `x`, a w nocy spada o `y`, to na dobę pokonuje `x-y`. Podobnie jak w pierwszym przykładzie zakładamy, że `1000` jest wielokrotnością `x-y`.

```
def kiedy(x, y):
    return 1000//(x-y)
```

Ostatecznie przechodzimy do rozwiązania właściwego zadania: żuczek w dzień wchodzi o `x`, a w nocy spada o `y`, a `1000` nie musi być wielokrotnością `x-y`.

```
def kiedy(x, y):
    return (1000-y-1)//(x-y)+1
```

Zachęcamy do przeanalizowania powyższego kodu i przetestowania go na kilku przykładach. Zadanie nie wymaga ani zaawansowanych umiejętności matematycznych, ani programistycznych, nie jest jednak proste. Samodzielne rozwiązanie jest doskonałym wyzwaniem dla początkującego programisty.

4. Wyzwanie 2 – Ile masz dni?

Zadanie

Napisz funkcję `imd(rok, miesiac, dzien)`, której wartością jest liczba dni, która upłynęła od danej daty do 27 czerwca 2017.

Przykłady:

Wynikiem funkcji dla `imd(1980, 12, 31)` jest `13327`,

wynikiem funkcji dla `imd(1999, 3, 11)` jest `6683`,

wynikiem funkcji dla `imd(2000, 1, 1)` jest `6387`.

Zadanie pochodzi z konkursu LOGIA 12 (etap 1).

Potrzebna wiedza

Uczeń powinien znać podstawowe operacje matematyczne oraz wiedzieć jak definiować funkcje. Ponadto przydatna będzie umiejętność korzystania ze zmiennych, znajomość instrukcji przypisania i warunkowej, a także podstawowych operacji logicznych.

```
x = 1 # x niech się stanie 1
```

```
print(x)
# x niech się stanie x+1, czyli zwiększ x o 1
x = x+1
print(x)

def czy_podzielna3(x):
    if (x%3 == 0):
          return True
    else:
          return False
```

Rozwiązanie

Dzielimy problem na podproblemy – mniejsze zadania, które rozwiążemy krok po kroku. Pierwszym podproblemem są lata przestępne. Rok przestępny to taki rok, który jest podzielny przez `4`, ale nie jest podzielny przez `100` lub jest podzielny przez `400`. Piszemy funkcję `czy_przestepny(rok)`, której wynikiem będzie `True` (prawda), gdy rok jest przestępny, `False` (fałsz) w przeciwnym przypadku.

```
def czy_przestepny(rok):
    if (rok%4 == 0):
          if ((rok%100 == 0) and (rok%400 != 0)):
                return False
          else:
                return True
    else:
          return False
```

Można to też zapisać inaczej nie stosując instrukcji warunkowej, ale wyrażenia logiczne.

```
def czy_przestepny(rok):
   return (rok%4==0) and (rok%100!=0) or (rok%400==0)
```

Następnym zadaniem jest zamiana pełnych lat na dni. Dokładniej liczymy, ile dni upłynęło od początku roku 1900. Gdy rok nie jest przestępny ma 365 dni, stąd `x = 365*(rok-1900)`. Dla roku przestępnego dodajemy 1.

```
def lata_nadni(rok):
    x = 365*(rok-1900)
```

```
    if (rok > 1900):
        p = (rok-1900-1)//4
    else:
        p = 0
    return x+p
```

Podobnie liczymy, ile dni upłynęło od 1 stycznia danego roku. Zapis jest tutaj bardziej skomplikowany, gdyż mamy miesiące, które mają 31 dni, 30 dni oraz luty 28 lub 29 dni.

```
def mies_nadni(rok, miesiac):
    ile = 30*(miesiac-1)
    ile = ile+miesiac//2
    if (miesiąc > 7): ile = ile+miesiac%2
    if (miesiac > 2): ile = ile-2
    if czy_przestepny(rok) and (miesiac > 2):
        ile = ile+1
    return ile
```

Pozostaje napisać treść funkcji głównej. Będzie ona wykorzystywać funkcje pomocnicze zdefiniowane wcześniej. Będziemy liczyć, ile dni upłynęło od 1 stycznia 1900 roku dla daty początkowej i końcowej.

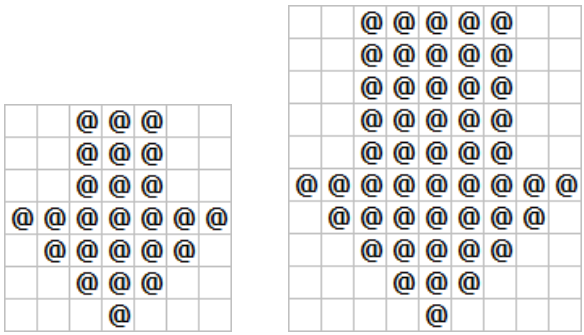
```
def imd(rok, miesiac, dzien):
    #data koncowa - 27 czerwca 2017
    data_do = lata_nadni(2017)+mies_nadni(2017, 6)+27
    data_od = lata_nadni(rok)+mies_nadni(rok,miesiac)
                                                +dzien
    return data_do-data_od
```

Ogólnie, jest to zadanie ciekawe dla nastolatków (starszych nie wypada pytać o wiek!) i niezwykle pouczające. Mimo że odnosi się do zagadnień z codziennego życia zrozumiałych dla każdego np. liczba dni w miesiącu, to formalny zapis nie jest prosty. Jedną z ważnych umiejętności, którą kształtujemy przy okazji tego zadania, jest zdolność zapisania w języku formalnym tego, czym posługujemy się na co dzień. Ponadto przy tym zadaniu, dla każdej napisanej funkcji powinniśmy opracować zestaw testów, który pozwolił się przekonać, że to co było intencją programisty, zostało faktycznie zapisane w języku programowania. Zachęcamy do samodzielnego zmierzenia się z tym wyzwaniem.

5. Wyzwanie 3 – Małpy i strzałka

Zadanie

Napisz funkcję `strzalka(n)`, po wywołaniu której będą wypisane na ekranie znaki `@` tworzące strzałkę. Trzonek strzałki tworzy kwadrat wielkości `n x n`, grot rozpoczyna `n+4` znaków, każda kolejna linijka ma 2 znaki mniej, kończy się jednym znakiem. Zakładamy, że parametr jest liczbą nieparzystą z zakresu od `3` - `49`.



Rysunek 1 Strzałki dla parametru `3` i `5`

Potrzebna wiedza

Uczeń powinien wiedzieć jak korzystać z pętli `for`, w której powtarzamy `n` razy ciąg instrukcji. Poniższa pętla będzie wykonana dla `i` od `0` do `4`, czyli `5` razy. Zmienna `i` jest nazywana zmienną sterującą pętlą.

```
for i in range(5):
    print("x")
```

Instrukcja `print("x")` wypisuje znak `x`, następnie przechodzi do następnej linii. Przydatna jest zmiana sposobu wypisywania, aby przejście do następnego wiersza nie nastąpiło po pojedynczej instrukcji `print`. Parametr `end` definiuje separator.

```
print("a", end = "!"); print("b")    wypisze a!b
print("a", end = ""); print("b")     wypisze ab
```

Rozwiązanie

Zacznijmy od rysowania trzonu. W każdym wierszu są wypisane 2 spacje, następnie `n` znaków `@`. Można to zapisać następująco:

```
print("  ", end = "")
```

```
for i in range(n):
    print("@", end = "")
print("")
```

Trzon składa się z `n` linii, dlatego należy zastosować jeszcze jedną pętlę `for`. Kod rysowania `n` linii złożonych z dwóch spacji i `n` znaków `@` wygląda następująco:

```
#trzon
for j in range(n):
    print("  ", end = "")
    for i in range(n):
        print("@", end = "")
    print("")
```

Powyższy kod można uprościć stosując możliwości Pythona w zakresie dodawania i mnożenia napisów.

```
for j in range(n):
    print(2*" "+n*"@")
```

Po przetestowaniu fragmentu kodu rysowania trzonu strzałki możemy przystąpić do rysowania grotu strzałki. Tutaj też mamy w każdej linii wypisywanie spacji i znaków `@`, liczba spacji zwiększa się o `1`, a liczba znaków `@` zmniejsza się o `2` w kolejnej linii. Znając już możliwości Pythona w dodawaniu i mnożeniu napisów, kod rysowania `p` spacji i `m` znaków `@` w linii wygląda następująco:

```
# m - liczba @, p - liczba spacji
print(p*" "+m*"@")
```

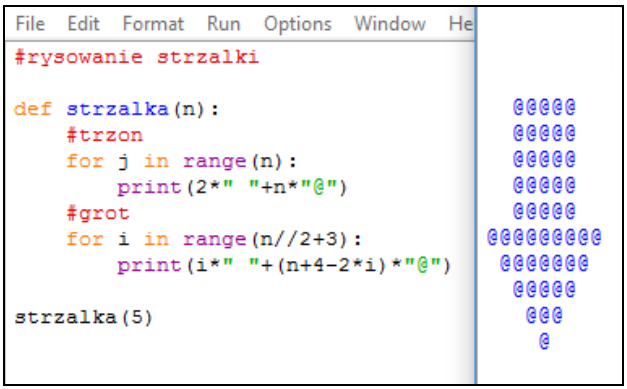
Musimy się zastanowić, jak wysoki jest grot strzałki. Dla `n = 3` wysokość wynosi `4`, dla `n = 4` wysokość wynosi `5`, `n = 5` wysokość wynosi `6` itp. Musimy znaleźć wzór na liczbę powtórzeń w pętli `for` rysowania grotu strzałki. Można zauważyć, że liczba ta wynosi `n//2+3`.

W pierwszej linii grotu liczba znaków `@` jest większa o `4` od wartości parametru `n`. W kolejnych liniach liczba znaków `@` zmniejsza się o `2`, a liczba spacji zwiększa się o `1`.

```
m = n+4
p = 0
for i in range(n//2+3):
    print(p*" "+m*"*")
```

```
p = p+1
m = m-2
```

Zamiast zmiennej `p` i `m` do rysowania spacji można wykorzystać zmienną i sterującą pętlą for.



Rysunek 2 Kod funkcji i wynik działania dla `n = 5`.

Kod programu rysowania strzałki jest bardzo prosty, nie zaczynamy jednak od niego, tylko analizując problem piszmy ten kod krok po kroku, jak powyżej.

6. Co warto wziąć pod uwagę przy nauce programowania

Naukę programowania warto połączyć z nauką algorytmiki oraz logicznego myślenia. Zadania dla uczniów powinny być ciekawe, trzeba pokusić się i przygotować treści zadań według ich zainteresowań, na przykład zamiast liczyć króliki w przypadku obliczania wartości n-tego elementu ciągu Fibonacciego można napisać treść zadania bardziej przyjazną uczniom. Na przykład:

Wojtek chcąc się zachęcić do nauki programowania wyznaczył sobie nagrody za każde prawidłowo rozwiązane zadanie. Za pierwsze zadanie otrzyma jedną owocową przekąską zwaną Misie frutisie, za kolejne zadanie również jedną owocową przekąską. Począwszy od trzeciego zadania liczba owocowych przekąsek będzie obliczana jako suma dwóch poprzednich wartości. Pomóż Wojtkowi napisać program obliczania, ile przekąsek otrzyma po rozwiązaniu n-tego zadania.

Ważny jest również efekt na ekranie, rysowanie z wykorzystaniem instrukcji `print` może być zabawą nawet dla mniej zainteresowanych programowaniem.

Należy także uważać, aby nie przesadzić z pisaniem programów z wykorzystaniem matematyki, żeby nie odbiegały one wiele od programu tego przedmiotu. Warto porozmawiać na ten temat z matematykiem i skorelować swoje działania.

7. Podsumowanie

Naukę programowania opieramy na rozwiązywaniu zadań algorytmicznych o wzrastającym stopniu trudności. Od pierwszych kroków przygotowujemy zadania, które nie tylko sprawdzają znajomość składni języka, ale wymagają myślenia. Do rozwiązania poszczególnych zadań, a co za tym idzie nauki programowania, potrzeba zarówno systematyczności, jak i wytrwałości. Uczeń, poznając nowy materiał, napotyka różne trudności, które musi przezwyciężyć. Chętniej będzie to robił, gdy zadania są ciekawe, związane z zainteresowaniami młodych ludzi.

Uczniowie, poznając tajniki programowania, mogą rozwijać umiejętność uczenia się. Uczą się planować rozwiązywanie zadania, dzielić problem na mniejsze problemy, które potrafią rozwiązać, implementować i testować swoje rozwiązania. Programowanie jest więc rozumiane szerzej niż tylko kodowanie. Podejście informatyczne zakłada przejście od specyfikacji problemu, poprzez znalezienie, i opracowanie i zaprogramowanie rozwiązania. Na koniec jest ono testowane. Tak rozumiane programowanie wspomaga kształcenie takich umiejętności jak: logiczne myślenie, precyzyjne prezentowanie myśli i pomysłów, sprzyja dobrej organizacji pracy, buduje kompetencje potrzebne do pracy zespołowej i efektywnej realizacji projektów (por. [6]). Myślenie w uporządkowany i ustrukturalizowany sposób jest przydatne w praktyce szkolnej i codziennym życiu.

Literatura

1. Borowiecki M., *Python na lekcjach informatyki w szkole ponadgimnazjalnej*, IwE 2013.
2. Jochemczyk W., Olędzka K., *Python dla wszystkich*, IwE 2016.
3. Jochemczyk W., Olędzka K., *Ważenie a system binarny*, IwE 2016.
4. Materiały dydaktyczne do nauki programowania w języku Python, http://python.oeiizk.edu.pl, ostatni dostęp 25.06.2017 roku.
5. Perekietka P., *Informatyka Unplugged (bez komputera) w kształceniu myślenia komputacyjnego*, IwE 2016.
6. Podstawa programowa kształcenia ogólnego dla szkoły podstawowej; Dziennik Ustaw Rzeczypospolitej Polskiej 2017, pozycja 356.
7. Sysło M., *Myślenie komputacyjne: nowe spojrzenie na kompetencje informatyczne,* IwE 2014.
8. Sysło M., *Wprowadzając ... porządek*, IwE 2016.