Sprawozdanie

Programowanie funkcyjne

Karolina Sikorska

Temat: *Gra karciana - Blackjack*

Wstępne zasady gry

Gracz nie widzi jednej z kart krupiera, co trzeba będzie uwzględnić podczas tworzenia „interfejsu" (w postaci wydruków) gry. Ograniczenia pola rozgrywki w postaci ograniczenia górnego >21, którego przekroczenie będzie skutkować zakończeniem gry na korzyść drugiego z uczestników, oraz ograniczenie dolne dla krupiera, gdy suma wartości jego kart nie przekracza 17 punktów (krupier musi zagrać). Gracz ma do wyboru dwie opcje – granie dalej jako dobranie dodatkowej karty do ręki, lub spasowanie i w konsekwencji przejście do instrukcji krupiera (bądź podsumowującej wynik gry, w zależności czy ręka krupiera w sumie ma >17 punktów). Wylosowane karty są na bieżąco usuwane z talii. As wylosowany przez którąkolwiek ze stron ma wartość 11 bądź 1, w zależności co jest dla uczestnika w danej sytuacji korzystniejsze – jeżeli suma jego kart przekracza, z Asem = 11, 21 to as wtedy As będzie liczony jako 1. Po zakończeniu rozgrywki gracz powinien dostać możliwość ponowienia jej bez konieczności ponownego uruchamiania skryptu. Możliwość zremisowania w momencie wystąpienia równości sum punktowych.

Rozwinięcie - funkcje

Pierwsze instrukcje (definiowane funkcje) *muszą* dotyczyć podstawy – puli kart, z której można będzie dobierać. Aby wszystko mogło się na bieżąco zapisywać trzeba zdefiniować zmienne lokalne w postaci list (i listy słowników), nie byłam w stanie znaleźć innego rozwiązania oferującego możliwość zapisu danych wynikowych w formacie jaki obrałam. Karty będą wyświetlać się zgodnie z obranym przeze mnie wzorem:

```
[{'Wartość karty':wartosc, 'Rodzaj':rodzaje}]
```

,gdzie rodzajem karty będzie jej kolor.

Pierwsza wersja kodu zakładała zakończenie funkcji w tym momencie, jednakże przy wydruku „interfejsu" zauważyłam powtarzalność kart, za każdym razem gdy rozgrywkę rozpoczęłam od nowa losowane były te same karty. Aby uniknąć takiego problemu zaimportowałam pakiet *random* i użyłam zawartej w nim funkcji *random.shuffle()*, która za każdym wywołaniem pierwszej funkcji będzie „tasować" talię. Zapewni to losowość w wyborach kart.

Mając podstawowe rodzaje kart i ich nazwy, w celach obliczeń, należy zdefiniować coś co pozwoli na dodawanie wartości kart do siebie (stworzenie funkcji sumującej). Dodawanie stringów nie jest dobrą opcją ponieważ będzie to po prostu wydłużać string (np. '1' + '2' = '12'), a w przypadku przekonwertowania wartości zawartych w liście *wartosci* na typ int()

problem pojawi się z kartami typu król, dama, itd.. Chciałam zachować konwencje znaną z kart - nie zostawiać damy, króla i waleta jako bezpośrednie wartości kart (=10). Z tego powodu potrzebne jest zdefiniowanie następnej zmiennej, która będzie słownikiem. Przypisze to do danych kart (keys) ich wartości (values) w postaci liczbowej.

Następną częścią jest stworzenie modelu dodawania kart. Pierwsza wersja skryptu zakładała dodawanie elementów na podstawie przypisanych do nich kluczy, jednakże w moim wykonaniu wyszło na to, że dla każdego dodania karty musiałaby istnieć dodatkowa funkcja umożliwiająca dodanie następnej. Pomysł może dobry, ale nie dla tego typu rozgrywki – jest mało uniwersalny. Prawdopodobieństwo tego, że gracz będzie losował ze zbioru kart same karty o wartości 2 jest bardzo niskie – ale istnieje, wtedy tych funkcji musiałoby być wiele, co utrudniałoby czytelność i funkcjonalność skryptu. Druga wersja zakładała mocno proceduralne dodawanie kart, a uwzględnienie Asa jako liczby 1/11 było osobną funkcją użytkową, ale po dłuższym przemyśleniu udało mi się skompilować kod za pomocą funkcji *sum()* i techniki *list comprehension*, żeby uniknąć dwóch funkcji napisanych stricte proceduralnie. Wygląda to bardziej elegancko i jest dostosowane pod koncepcje programowania funkcyjnego (mimo że nie zgadza się z nim w 100% [obecność instrukcji warunkowych]). W tej samej funkcji uwzględniam obecność Asa – jeżeli karta znajduje się w ręce gracza/krupiera to musi przyjąć wartość bardziej korzystną – jeżeli suma kart jest mniejsza bądź równa 11 (gdy as = 1, dlatego tak też jest zdefiniowany w słowniku wartości kart) to do sumy kart zostanie dodane 10. Jeżeli suma kart przekracza 11 to As przyjmie wartość 1, aby nie dopuścić do przekroczenia 21 punktów przez gracza/krupiera.

W tym momencie w kodzie mamy funkcje odpowiadającą za samą talię, wartości kart w talii oraz sumowanie. Do całości trzeba zrobić prosty interfejs graficzny w postaci wydruków, w konsoli, oraz opcję dobierania kart przez obie ze stron w danych wypadkach (u krupiera trzeba uwzględnić warunek suma<17).

Dla gracza wystarczy prosta funkcja, która będzie usuwać z talii kartę i dodawać ją (na podstawie dodawania elementu do listy) do ręki gracza. Przy okazji chciałam dodać aspekt graficzny, przy dobieraniu przez gracza karty, w postaci wydruku. Wspomniany aspekt kodu również przysporzył podczas tworzenia małego problemu – korzystałam wtedy tylko z jednej funkcji dobierania dla obu uczestników, więc kiedy krupier dobierał kartę to również wyświetlało się to, w postaci wydruku, w konsoli, jako „Dobieram kartę…" - w teorii zgodnie z konwencją, ale utrudniało czytelność wydruku, a poza tym to krupier dobiera kartę, a nie uczestnik, więc chcąc być w 100%. poprawnym wydruk winien wyglądać następująco: „Krupier dobiera kartę…". Zastosowanie czegoś takiego znowu kierowałoby na użycie instrukcji warunkowych, czego chciałam uniknąć – stąd stworzenie dwóch osobnych funkcji dla dobierania kart.

W przypadku dobierania karty przez krupiera trzeba uwzględnić warunek mówiący, że krupier ma dobierać karty do momentu uzyskania sumy>17. Żeby ponownie uniknąć stosowania warunków, zdecydowałam się na pętle while, która będzie (w ten sam sposób jak w przypadku dobierania karty przez gracza) usuwać kartę z puli oraz dodawać ją do talii krupiera. Pętla będzie się powtarzać do momentu uzyskania przez krupiera sumy wartości kart większej od 17.

Ostatnim z elementów podstawowych rozgrywki będzie stworzenie podstawowego interfejsu graficznego w postaci wydruków. Podstawowa zasada gry zakłada, że gracz zna tylko jedną z dwóch kart krupiera przed podsumowaniem rozgrywki, dlatego przy

budowaniu funkcji zastosowałam dodatkowy argument ‘pokaz_karty’, który ma przypisaną wartość domyślną typu bool =False. Oznacza to, że jeśli argument ten nie zostanie przekazany podczas wywoływania funkcji, zostanie użyta ta domyślna wartość. W celu identyfikacji tej wartości trzeba zastosować instrukcję warunkową, inaczej funkcja nie byłaby w stanie stworzyć dwóch osobnych wydruków dla talii krupiera. Jeżeli zostanie przekazany argument trzeci dla funkcji to zamiast ukróconej talii krupiera wyglądającej następująco:

```
Talia krupiera: [{PierwszaKarta}, ‘X’]
```

Otrzymamy pełną wersję talii krupiera razem z sumą jego punktów:

```
Talia krupiera: [{PierwszaKarta}, …, {N-taKarta}]
Suma punktów: SumaPunktówKrupiera
```

Ostatnim krokiem do stworzenia gry będzie utworzenie głównego „mechanizmu” funkcjonalnego gry, który będzie osobną funkcją opierającą swoje działanie na utworzonych poprzednio funkcjach.

Pierwszym pomysłem w tworzeniu głównego mechanizmu był kompletny proceduralny twór złożony wyłącznie z instrukcji warunkowych, co bardzo minęłoby się z założeniem programowania funkcyjnego. Pomimo prób redukcji ilości instrukcji warunkowych niestety większość z nich pozostała – w inny sposób nie jestem w stanie sprawdzić prawdziwości testowanych stwierdzeń. Główny mechanizm opiera się na zagnieżdżeniu funkcji while. *While True*, pozwala na wykonywanie podanych pod sobą instrukcji do momentu zaprzeczenia. Zagnieżdżenie funkcji będzie umożliwiało późniejsze ponowienie rozgrywki, jeżeli nie uwzględniałabym możliwości ponownego wykonania skryptu pojedyncza funkcja *while true*, powinna wystarczyć do poprawnego wykonania programu.

„Głowica” gry (root())

Najpierw należy rozdać graczowi i krupierowi karty. Tworzymy nową talię (opierając się na stworzonej na początku funkcji), z której będziemy usuwać elementy, które następnie znajdą się na liście, tworzącej rękę uczestnika:

```
gtalia = talia()
gracz = [gtalia.pop(), gtalia.pop()]
krupier = [gtalia.pop(), gtalia.pop()]
```

Będzie to reprezentować pierwsze rozdanie kart, a jeżeli chodzi o aspekt graficzny, po wstawieniu do wzoru zawartego w funkcji tworzącej interfejs, otrzymamy wydruk:

```
Twoja talia: gracz
Suma punktów: sumuj(gracz)
```

Na tym samym poziomie „zaawansowania” musimy dodać opcję ponownego zagrania bez wyłączenia skryptu, po zakończonej rozgrywce, w tym celu poza głowicą gry trzeba stworzyć następną funkcję. Funkcja ta będzie pytać użytkownika, który jeżeli odpowie twierdząco – zapętli mechanizm gry, a jeżeli odpowie w każdy inny sposób to zainicjuje wyjście z pętli zewnętrznej. Dzięki zastosowaniu .strip(), przy odpowiedzi użytkownika nie będą brane pod uwagę wszelkie znaki białe takie jak tabulatory czy spacje. Co prawda takie rozwiązanie umożliwia użytkownikowi wprowadzenie innej wartości niż Tak bądź Nie (tutaj: Y/N), żeby uniknąć ponownego zastosowania instrukcji warunkowej każda inna odpowiedź od Tak

(tutaj: Y) będzie traktowana jako brak wyrażenia chęci do ponownego rozegrania i automatyczne zatrzymanie pracy skryptu.

Zgodnie z regułą najpierw wykonywane są instrukcje w pętli zewnętrznej poprzedzające pętlę wewnętrzną, potem to co znajduje się w pętli wew. a następnie reszta kodu, z tego faktu korzystam właśnie przy definicji ponownego rozpoczęcia rozgrywki.

Pętla wewnętrzna na początku powinna wyświetlić wydruk z utworzonym wcześniej interfejsem wzbogaconym o wcześniej wylosowane talie, przez gracza i krupiera, w pętli zewnętrznej. W tym momencie gracz może już osiągnąć pulę 21. punktów, przez wylosowanie asa i króla/damy/waleta/10., stąd trzeba postawić pierwszy warunek, którego wynikową będzie opuszczenie pętli wewnętrznej i przejście do funkcji pytania o ponowną rozgrywkę. Umieszczenie tego warunku na samym początku będzie prowokowało sprawdzanie go jako pierwszego przy każdym obrocie pętli.

Jeżeli jednak gracz w pierwszym rozdaniu nie osiągnie 21. punktów, zgodnie z regułami gry musi zostać postawiony przed wyborem – kontynuacja rozgrywki i dobranie karty bądź spasowanie i przejście do kolejki krupiera. Jeżeli gracz zdecyduje się na dobranie karty aktywuje to stworzoną wcześniej funkcję dobierania karty dla gracza. W tym momencie również trzeba uwzględnić możliwość przekroczenia 21. punktów, wtedy powinien zostać wyświetlony interfejs zawierający dodatkowo wszystkie karty krupiera, jak i również sumę jego punktów. Tutaj przyda się stworzona wcześniej funkcja, w której zastosowałam przyjęcie wartości jednego z jej argumentów jako False. Aby wszystko zostało wyświetlone prawidłowo przy wywołaniu funkcji definiujemy argument pokaz_karty jako True i generujemy przy tym odpowiadający wydruk:

```
                    if sumuj(gracz) > 21:
            interfejs(gracz, krupier, pokaz_karty=True)
    print('\nSuma twoich punktów przekroczyła 21. Krupier wygrywa.')
```

Jeżeli jednak gracz zdecyduje się na przerwanie rozgrywki, trzeba przejść do ręki krupiera. Na początku wywołujemy stworzoną wcześniej funkcję dobierania dla krupiera. Dzięki temu, że zdecydowałam się na ujęcie w niej pętli while, sprawdzającej czy krupier ma w ręce sumę kart <17, jeżeli po pierwszym rozdaniu krupier będzie miał w talii sumę większą niż 17 to tak naprawdę funkcja nie zostanie wykonana, a główny mechanizm przejdzie do wykonania kolejnej instrukcji.

Wywołam pełen interfejs gry, żeby następnie przejść do podsumowania. Podsumowaniem będzie następna instrukcja warunkowa. Oprócz opcji wygranej/przegranej dla obu stron zdecydowałam się na stworzenie opcji remisowej. W tym celu zdefiniuje prostą funkcję która zwraca równość sum wartości kart gracza do krupiera. Jeżeli warunek będzie spełniony to wywołany zostanie wydruk, a następnie opuszczona zostanie pętla wewnętrzna dzięki instrukcji break. Jeżeli nie będzie równości punktowej, sprawdzony zostanie warunek mówiący o zwycięzcy – skoro gracz nie ma blackjacka, nie przekroczył 21. punktów, ani nie ma remisu punktowego, to jedna ze stron musiała wygrać.

Tworzymy prostą zmienną, która w zależności od tego czy suma gracza jest większa od sumy krupiera bądź krupier przekracza 21. punktów, ma przypisaną wartość typu string „Gracz”, w innym przypadku wartość typu string „Krupier”. Całość wywołuje instrukcja print, aby zgadzało się to z przyjętą koncepcją prostego, czytelnego interfejsu graficznego.

Korzystając z tego, że i tak byłam zmuszona do użycia instrukcji warunkowych w głównym bloku gry, zdecydowałam się na dodanie „grzecznościowej” opcji, w przypadku popełnienia literówki przez gracza, przy możliwości dobrania karty/spasowania. Nie zawarłam w niej instrukcji break, więc dojdzie do ponownego obrotu pętli.

Aby nie wywoływać ręcznie funkcji głównej gry (root()) na końcu dodałam specyficzny dla języka Python idiom, który inicjuje kod w momencie bezpośredniego uruchomienia skryptu. Pozwala to na importowanie modułu do innych skryptów, jednocześnie pozwalając na inicjacje kodu tylko wtedy gdy plik skryptu zostanie uruchomiony bezpośrednio. Zapewni to pewnego rodzaju elastyczność i możliwość wykorzystania kiedyś w innych skryptach.

Podsumowanie

Pomimo prób ograniczenia praktyk programowania imperatywnego, nie udało mi się uniknąć definiowania zmiennych lokalnych bądź też używania instrukcji warunkowych. Najprawdopodobniej istnieje opcja stworzenia podobnie działającego skryptu, w taki sposób aby był on bardziej zgodny z koncepcją programowania funkcyjnego, jednakże wtedy, (najprawdopodobniej) nie byłoby opcji zawarcia części elementów, które uznałam za stosowne w momencie tworzenia części ideowej mojego programu (bieżących wydruków, pamiętania wartości itp.). Trzeba byłoby zastosować inne metody wybierania wartości kart, np. ograniczenie się tylko do talii z wartościami (pomnożonej razy 4, ze względu na 4 istniejące kolory), bez uwzględniania koloru jako osobnej zmiennej.

Źródła/bibliografia

Kod oraz sprawozdanie stworzone na podstawie wiedzy własnej, informacji z wątków na serwisie **StackOverflow** oraz poniższej literatury:
- *Myśl w języku Python!* - **Allen B. Downey**
- *Python. Wprowadzenie* – **Mark Lutz**