Cvičení 3, skupiny 06,07:

2.3 ze sbírky, 3.1 ze sbírky, pro rychlé: převod na binární čísla, výpočet odmocniny

Procvičování na umimeinformatiku.cz

Na přednášce se dosud objevila hlavně témata:

- https://www.umimeinformatiku.cz/programovani-promenne-a-ciselne-vyrazy
- https://www.umimeinformatiku.cz/programovani-podmineny-prikaz-if
- https://www.umimeinformatiku.cz/programovani-cyklus-for
- https://www.umimeinformatiku.cz/programovani-cyklus-while

Fibonacciho posloupnost – řešený příklad

Naprogramujeme si funkci, která vypisuje prvních *n* členů Fibonacciho posloupnosti. První dva členy této posloupnosti jsou rovny 1 a potom je každý následující člen určen jako součet dvou předchozích členů. Začátek posloupnosti vypadá takto: 1 1 2 3 5 8…

Řešení používá pomocnou proměnnou *temp*, do které si uložíme hodnotu aktuálního prvku posloupnosti *current* předtím, než ji updatujeme, protože tuto hodnotu nesmíme přiřazením `current = next` přepsat a zapomenout, ještě ji budeme potřebovat k updatu hodnoty *next*.

```
def fibonacci(n):
    current = 1
    next = 1
    for i in range(n):
        print(current, end=" ")
        temp = current
        current = next
        next = temp + next
```

Vyzkoušejte volání:

```
fibonacci(7)
```

Výstup by měl být:

```
1 1 2 3 5 8 13
```

Textová grafika

Vykreslování různých obrazců do výstupu pomocí `print`. Nejdříve zkusme pomocí dvou vnořených for-cyklů vypsat čtverec velikosti *n* pomocí znaků `#`.

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

Zkuste si, jaký efekt má na podobu výsledného čtverce použití `end=" "` nebo `end=""`.

Naprogramujte si další příklady ze sekce 2.3 ze sbírky (řešení pro Prázdný čtverec, Pyramidu a Písmeno N najdete na konci tohoto dokumentu, ale vyzkoušejte si nejdřív naprogramovat sami).

Jednoduché výpočty

V tuto chvíli začneme ve funkcích používat i příkaz `return`. Jaký je rozdíl mezi `print` a `return`?

`print` – používali jsme dosud, funkce něco vypočítá, vypíše na obrazovku a skončí.

`return` – funkce skončí, jakmile provede příkaz `return` a vrací vypočítanou hodnotu (kterou můžeme dál použít na místě, kde funkci voláme).

Příklad funkce, která vrací o 15 % zvětšené číslo:

```
def plus_dph(price):
    return 1.15 * price
```

Příklady použití:

`print(plus_dph(10))` vypíše 11.5

`final_price = plus_dph(10)` přiřadí spočítanou hodnotu 11.5 do proměnné `final_price`

```
if a <= plus_dph(10):
    print(a)
```

Výsledek volání funkce můžeme použít stejně jako jakoukoli jinou hodnotu např. v podmínce, parametr při volání jiné funkce, atd.

Příklad volání, které nemá smysl:

```
plus_dph(10)
```

Toto samo o sobě nic nevypíše, ani jsme si spočítanou hodnotu tímto voláním nikam neuložili.

Příklad s faktoriály

Chceme vypsat prvních *n* členů posloupnosti faktoriálů přirozených čísel, tj. začínáme od 1.
(Faktoriál *n!* je součin čísel 1,2,...*n*, viz přednáška.)
Použijeme pomocnou funkci `factorial(k)` pro výpočet faktoriálu.

```
def factorial(k):
    value = 1
    while k > 0:
        value *= k
        k -= 1
    return value

def output_factorials(n):
    for i in range(1, n + 1):
        print(factorial(i), end=" ")
```

`output_factorials(5)` pak vypíše `1 2 6 24 120`

Příklad s cifernými součty

Hledáme číslo menší než N, které má největší ciferný součet (je jednoznačné?). Pomocná funkce – ciferný součet `digit_sum(n)`.

```
def digit_sum(n):
    soucet = 0
    while n > 0:
        soucet += n % 10
        n = n // 10
    return soucet

def max_digit_sum(N):
    sum_max = 0
    result = 0
    for i in range(1, N):
        if digit_sum(i) > sum_max:
            sum_max = digit_sum(i)
            result = i
    print("Number", result, "has maximal digit sum.")
```

Funkce `max_digit_sum` prochází v cyklu čísla do N a porovnává jejich ciferné součty s dosud nalezeným největším ciferným součtem. V `sum_max` si uchovává maximální ciferný součet a v `result` číslo, které má ciferný součet `sum_max`.

Jak je to s jednoznačností takového čísla? (Například pro N=10 existuje jediné číslo 9 s maximálním ciferným součtem 9. Ale pro N=19 existují dvě čísla 9 a 18 se stejným maximálním ciferným součtem 9.) Upravíme program, přidáme boolovskou proměnnou `unique`:

```
def digit_sum(n):
    soucet = 0
    while n > 0:
        soucet += n % 10
        n = n // 10
    return soucet

def max_digit_sum(N):
    sum_max = 0
    result = 0
    unique = True
    for i in range(1, N):
        if digit_sum(i) > sum_max:
            sum_max = digit_sum(i)
            unique = True
            result = i
        elif digit_sum(i) == sum_max:
            unique = False
    print("Number", result, "has maximal digit sum.")
    if unique:
        print("This number is unique.")
    else:
        print("Not unique.")
```

Jednoznačnost je pravdivá v okamžiku nalezení většího ciferného součtu, než byly všechny předtím. Jednoznačnost přestává být v pravdivá, když najdeme jiné číslo se stejným ciferným součtem.

Bonusová otázka: Současný výpočet funkce `max_digit_sum(N)` není až tak efektivní, v každé iteraci třikrát volá funkci `digit_sum`. Jak bychom mohli výpočet zrychlit?

Samostatné programování

Naprogramujte si několik funkcí s výstupem ze sekce 3.1 sbírky. Nejdříve zkuste naprogramovat sami. Potom pro kontrolu řešení prvního příkladu (součet čísel od 1 do *n*) najdete přímo ve sbírce, faktoriály a ciferný součet se dají dohledat ve slidech z přednášky.

Kdo si chce vyzkoušet naprogramovat převod čísla na binární zápis nebo výpočet odmocniny, může :) (kontrolu lze opět provést prozkoumáním slidů z přednášky).

Textová grafika – řešení

Prázdný čtverec

```
def empty_square(n):
    for i in range(n):
        for j in range(n):
            if i == 0 or j == 0 or i == n - 1 or j == n - 1:
                print("#", end=" ")
            else:
                print(".", end=" ")
        print()
```

Stejně jako u prvního příkladu na vykreslení celého čtverce pomocí # odpovídá *i* řádkům a *j* sloupcům. Uvnitř vnitřního for-cyklu pro výpis jednoho řádku (`for j in range(n)`) rozhodujeme pro každou pozici, zda jsou souřadnice *i,j* na okraji čtverce (píšeme #) nebo uvnitř (píšeme .).

Pyramida

```
def pyramid(n):
    for i in range(n):
        for j in range(2 * n - 1):
            if j >= n - 1 - i and j <= n - 1 + i:
                print('#', end=" ")
            else:
                print(' ', end=" ")
        print()
```

Vypisujeme jednotlivé řádky (for-cyklus přes řádky s proměnnou *i*). Jedním voláním `print` vytiskneme znak křížek nebo mezeru a pak ještě příslušný znak end, proto je výsledná pyramida tak trochu roztažená (v zadání bylo, že chceme roztaženou, takže v pořádku).
Na každém řádku provádíme *(2n-1)*krát `print`. Uprostřed řádku jsme pro *j=n-1.*

Na začátku řádku vytiskneme *(n-1-i)*krát mezeru, uprostřed vytiskneme *(2i-1)* znaků #, následuje opět *(n-1-i)*krát tisk mezer.

Velké písmeno N

```
def letter_N(size):
    height = max(3, size)
    width = height
    for row in range(height):
        for col in range(width):
            if col == 0 or col == width - 1 or row == col:
                print('#', end=" ")
            else:
                print(' ', end=" ")
        print()
```

Nejprve nastavíme výšku a šířku obrazce. Řádky a sloupce jsou potom označené *row* a *col*.