resitev

January 28, 2024

0.1 Prva funkcija

FRI ima, kot gotovo veste, pogodbe z različnimi uglednimi tujimi organizacijami (NSA, MI5, Mossad - you name it) za izvajanje občasnih del. Nekoč je ena od njih prestregla tole funkcijo v Pythonu. No, pravzaprav ne funkcije, temveč že prevedeno kodo te funkcije. Zanimalo jih je, kaj počne, kaj računa, kaj izpiše, kaj vrne (če kaj). Torej?

```
Eni pravijo, naj bi bil `n` menda argument funkcije. Pa še tega ne smemo pozabiti, da funkcija
```

```
  2           0 LOAD_CONST               1 (1)
              2 DUP_TOP
              4 STORE_FAST               1 (a)
              6 STORE_FAST               2 (b)

  3     >>    8 LOAD_FAST                0 (n)
             10 LOAD_CONST               2 (0)
             12 COMPARE_OP               4 (>)
             14 POP_JUMP_IF_FALSE       48

  4          16 LOAD_GLOBAL              0 (print)
             18 LOAD_FAST                1 (a)
             20 CALL_FUNCTION            1
             22 POP_TOP

  5          24 LOAD_FAST                2 (b)
             26 LOAD_FAST                1 (a)
             28 LOAD_FAST                2 (b)
             30 BINARY_ADD
             32 ROT_TWO
             34 STORE_FAST               1 (a)
             36 STORE_FAST               2 (b)

  6          38 LOAD_FAST                0 (n)
             40 LOAD_CONST               1 (1)
             42 INPLACE_SUBTRACT
             44 STORE_FAST               0 (n)
             46 JUMP_ABSOLUTE            8
        >>   48 LOAD_CONST               0 (None)
             50 RETURN_VALUE
```

0.1.1 Razkrivanje

Naloga je že povedala, da gre za funkcijo z argumentom `n`. Čeprav še ne vemo, kako v Pythonu pisati funkcije (pa tudi zank pravzaprav še ne poznamo), kar napišimo

```
def f(n):
```

Začetek je preprost: Python na sklad naloži konstanto 1, podvoji zadnji element sklada (tako da sta na skladu dve enici) in shraniti tadva elementa v `a` in `b`.

```
  2           0 LOAD_CONST               1 (1)
              2 DUP_TOP
              4 STORE_FAST               1 (a)
              6 STORE_FAST               2 (b)
```

To je prevod prirejanja `a = b = 1`.

```
def f(n):
    a = b = 1
```

Potem pa se stvari malo zavozlajo. Točneje, zazankajo. Imamo

```
  3     >>    8 LOAD_FAST                0 (n)
             10 LOAD_CONST               2 (0)
             12 COMPARE_OP               4 (>)
             14 POP_JUMP_IF_FALSE       48
```

To na sklad naloži vrednost `n` (kjer je `n` argument funkcije, saj mu očitno nihče ni priredil vrednosti) in `0`, ter ju primerja. Če ni res (`POP_JUMP_IF_FALSE`) je da `n > 0`, skoči na bajt 48. Torej nekaj preskoči. To bo najbrž `if` ali, morda, `while`. Dilemo razreši vrstica pred bajtom 48:

```
             46 JUMP_ABSOLUTE            8
```

Tu piše, da skočimo na bajt 8, torej ravno na začetek gornjega bloka. Gre torej za zanko: vse med bajtoma 8 in 46 (ali, če hočete, med 16 in 46) se ponavlja, dokler je `n > 0`. Imamo torej

```
def f(n):
    a = b = 1
    while n > 0:
```

Sledi

```
  4          16 LOAD_GLOBAL              0 (print)
             18 LOAD_FAST                1 (a)
             20 CALL_FUNCTION            1
             22 POP_TOP
```

To na sklad postavi funkcijo `print` in vrednost `a` ter pokliče to funkcijo (pri čemer ji pošlje en argument - to je pomen enice v `CALL_FUNCTION 1`). Vsaka funkcija v Pythonu vrne rezultat, torej ga vrne tudi `print`. Ker ga ne potrebujemo, ga vržemo stran (`POP_TOP` zavrže zadnji element sklada).

Prišli smo torej do

```
def f(n):
    a = b = 1
```

```
    while n > 0:
        print(a)
```

Sledi najbolj zapletena reč.

```
  5           24 LOAD_FAST                2 (b)
              26 LOAD_FAST                1 (a)
              28 LOAD_FAST                2 (b)
              30 BINARY_ADD
              32 ROT_TWO
              34 STORE_FAST               1 (a)
              36 STORE_FAST               2 (b)
```

Na sklad naloži `b`, nato `a`, nato spet `b`. Sklad je torej takšen: `b a b`. Nato sešteje zadnja dva elementa; sklad bo torej `b a + b`. Nato zamenja vrstni red zadnjih dve elementov, torej imamo `a + b b`. Potem shrani zadnji element v `a`; `a` bo torej enak `b`-ju in na skladu ostane `a + b`. Tega shrani v `b`.

Naivno bi torej to prepisali v

```
a = b
b = a + b
```

vendar gornja bajtkoda ne dela tega. V `b` se shrani vsota *prejšnjega* `a`-ja in `b`-ja, tule pa bi se v `b` v bistvu shranil `b + b`. Tisti, ki še ne znajo Pythona, bi tole lahko imitirali le s tretjo spremenljivko, recimo tako

```
t = a + b
a = b
b = t
```

Kdor zna Python (in ga zna dovolj dobro), pa ve, da je možno v njem prirejati tudi sočasno,

```
a, b = b, a + b
```

Kaj se skriva za tem in kako to v resnici deluje, bomo izvedeli prihodnji teden.

Prišli smo torej do

```
def f(n):
    a = b = 1
    while n > 0:
        print(a)
        a, b = b, a + b
```

Sledi

```
  6           38 LOAD_FAST                0 (n)
              40 LOAD_CONST               1 (1)
              42 INPLACE_SUBTRACT
              44 STORE_FAST               0 (n)
```

ki naloži `n` in `1`, ju odšteje in to shrani nazaj v `n`, se pravi `n = n - 1` ali `n -= 1`.

```
def f(n):
    a = b = 1
```

```
    while n > 0:
        print(a)
        a, b = b, a + b
        n = n - 1
```

Sledi skok, ki je del `while` zadnji dve vrstici,

```
        >>   48 LOAD_CONST               0 (None)
             50 RETURN_VALUE
```

pa poskrbita, da funkcija vrne `None`, ker vsaka funkcija v Pythonu pač nekaj vrne. Tidve vrstici se pojavita, četudi v funkcijo ne napišemo eksplicitnega `return None`.

Kaj počne funkcija, prepoznajo vsi, ki so že kdaj napisali funkcijo, ki računa Fibonacijeva števila. Računa namreč Fibonacijeva števila, zaporedje 1 1 2 3 5 8 13 21 34 55 …, v katerem je vsak člen enak vsoti prejšnjih dveh.

0.2 Druga funkcija

Če ste, draga študentka ali študent, potencialni kandidat za občasna pogodbena dela na FRI, se boste morali izkazati še z nekoliko težjo funkcijo. Kaj izračuna in vrne tale?

```
Videti je, da naj bi bila `n` in `x` argumenta funkcije, saj jima nihče ne prireja vrednosti.
```

```
  2           0 LOAD_CONST               1 (1)
              2 STORE_FAST               2 (r)

  3     >>    4 LOAD_FAST                1 (n)
              6 LOAD_CONST               2 (0)
              8 COMPARE_OP               4 (>)
             10 POP_JUMP_IF_FALSE       50

  4          12 LOAD_FAST                1 (n)
             14 LOAD_CONST               3 (2)
             16 BINARY_MODULO
             18 LOAD_CONST               1 (1)
             20 COMPARE_OP               2 (==)
             22 POP_JUMP_IF_FALSE       32

  5          24 LOAD_FAST                2 (r)
             26 LOAD_FAST                0 (x)
             28 BINARY_MULTIPLY
             30 STORE_FAST               2 (r)

  6     >>   32 LOAD_FAST                0 (x)
             34 LOAD_FAST                0 (x)
             36 BINARY_MULTIPLY
             38 STORE_FAST               0 (x)

  7          40 LOAD_FAST                1 (n)
             42 LOAD_CONST               3 (2)
```

```
             44 BINARY_FLOOR_DIVIDE
             46 STORE_FAST               1 (n)
             48 JUMP_ABSOLUTE            4

  8     >>   50 LOAD_FAST                2 (r)
             52 RETURN_VALUE
```

0.2.1 Razvozlavanje

En kos hitro prepoznamo. Začetnemu prirejanju, `r = 1` sledi zanka `while n > 0`, tako kot v prejšnjem programu. Imamo torej

```
def f(x, n):
    r = 1
    while n > 0:
```

Sledi

```
  4          12 LOAD_FAST                1 (n)
             14 LOAD_CONST               3 (2)
             16 BINARY_MODULO
             18 LOAD_CONST               1 (1)
             20 COMPARE_OP               2 (==)
             22 POP_JUMP_IF_FALSE       32
```

Na sklad naloži `n` in 2, izračuna ostanek po deljenju z 2 (po čemer na skladu ostane ta ostanek); na sklad naloži 1 in primerja, če sta enaka. Z drugimi besedami, preveri, ali velja `n % 2 == 1`. Če to ni res, skoči na bajt 40. Tu bi lahko spet šlo za `while`, vendar imamo tokrat `if`, kar vidimo po tem, da v bajtu 40 oziroma tik pred njim ni nobenega skoka nazaj. Tale `POP_JUMP_IF_FALSE` samo preskoči naslednji blokec, naslednjo vrstico programa.

```
def f(x, n):
    r = 1
    while n > 0:
        if n % 2 == 1:
```

Ta naslednja vrstica, se je prevedla v

```
  5          24 LOAD_FAST                2 (r)
             26 LOAD_FAST                0 (x)
             28 BINARY_MULTIPLY
             30 STORE_FAST               2 (r)
```

in zdaj to menda že znamo prebrati: naloži `r` in `x`, ju zmnoži in rezultati shrani nazaj v `r`, torej

```
def f(x, n):
    r = 1
    while n > 0:
        if n % 2 == 1:
            r = r * x
```

Nato - in to je že izven `if`-a, saj je skok `POP_JUMP_IF_FALSE` skočil natančno sem - se zgodi

```
  6     >>   32 LOAD_FAST                0 (x)
             34 LOAD_FAST                0 (x)
             36 BINARY_MULTIPLY
             38 STORE_FAST               0 (x)
```

kar je `x = x * x`,

```
def f(x, n):
    r = 1
    while n > 0:
        if n % 2 == 1:
            r = r * x
        x = x * x
```

Sledi

```
  7          40 LOAD_FAST                1 (n)
             42 LOAD_CONST               3 (2)
             44 BINARY_FLOOR_DIVIDE
             46 STORE_FAST               1 (n)
```

To naloži na sklad `n` in 2, ju celoštevilsko deli in rezultat shrani nazaj v `n`, se pravi

```
def f(x, n):
    r = 1
    while n > 0:
        if n % 2 == 1:
            r = r * x
        x = x * x
        n = n // 2
```

Nato imamo skok, ki zaključi zanko. Sledi ji le še

```
  8     >>   50 LOAD_FAST                2 (r)
             52 RETURN_VALUE
```

Naložimo vrednost `r` in jo vrnemo. Celotna funkcija je torej

```
def f(x, n):
    r = 1
    while n > 0:
        if n % 2 == 1:
            r = r * x
        x = x * x
        n = n // 2
    return r
```

ali, če imamo raje inkrementalne operatorje

```
def f(x, n):
    r = 1
    while n > 0:
        if n % 2 == 1:
            r *= x
```

```
        x *= x
        n //= 2
    return r
```

0.2.2 Kaj počne ta funkcija?

Tole je pomembna funkcija. Uporabljamo jo stalno, ne da bi vedeli, da jo. Za začetek jo preskusimo s par argumenti in poskusimo uganiti, kar pravzaprav računa.

[1]:
```
def f(x, n):
    r = 1
    while n > 0:
        if n % 2 == 1:
            r *= x
        x *= x
        n //= 2
    return r
```

[2]:
```
f(2, 5)
```

[2]: 32

[3]:
```
f(5, 2)
```

[3]: 25

[4]:
```
f(10, 4)
```

[4]: 10000

[5]:
```
f(10, 6)
```

[5]: 1000000

[6]:
```
f(5, 3)
```

[6]: 125

Stvar je očitna: funkcija vrne x^n, pri čemer mora biti `n` celo število (ker celoštevilsko deljenje in tako naprej).

Funkcije, ki izračuna x^n, ni težko napisati niti, če je `while` edina zanka, za katero vemo.

[7]:
```
def f(x, n):
    i = 0
    r = 1
    while i < n:
        r *= x
        i += 1
```

```
    return r
```

Ali, če hočemo z zanko `for`, ki je še ne poznamo:

[8]:
```
def f(x, n):
    r = 1
    for i in range(n):
        r *= x
    return r
```

[9]:
```
f(5, 3)
```

[9]:
```
125
```

To deluje čisto dobro, dokler je n majhen. Če je n enak, recimo, `18446744073709551557`, pa bo računanje x^n se pravi $x^{18446744073709551557}$ trajalo kar dolgo, saj imamo zanko, ki se mora obrniti 18446744073709551557-krat. In to je pravzaprav sorazmerno majhen n (tole je, mimogrede, največje 64-bitno praštevilo). V praksi pogosto računamo potence s 1024-bitnim eksponentom, torej potence s skoraj tako velikim eksponentom, kot je

[10]:
```
2 ** 1024
```

[10]:
```
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216
```

Ko bi pognali `for i in range(179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216)` bi čakali in čakali in potem bi se naveličali čakati, in potem bi vesolje čakalo in čakalo in se naveličalo čakati in se končalo, kakorkoli se že ima namen končati … zanka pa bi tekla še naprej.

Očitno potrebujemo hitrejši način za računanje takšnih gromozanski potenc. Preden ga spoznamo, pa moramo razčistiti dvoje.

1. Zakaj potrebujemo take ogromne potence?
2. Če je to samo eksponent in torej računamo, na primer $2^{179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393}$ - mar ni to število preveliko, da bi ga shranili v pomnilnik?

Najprej odgovorimo na drugo vprašanje: da to število ima 179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216 bitov in je preveliko, da bi ga shranili v vesolje, ne samo v pomnilnik. Vesolje ima namreč samo kakšnih

[11]:
```
10 ** 72
```

[11]:
```
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

atomov.

Odgovor na drugo vprašanje je: na srečo nas ne zanima to število, temveč ostanek po deljenju tega števila z nekim številom. Ta ostanek je pač samo toliko veliko kot to, drugo število. Recimo, da imam

[12]:
```
x = 352786429304876293586734987629384 67
n = 
  ↪17976931348623159077293051907890247336179769789423065727343008115773267580550096313270847 73
m = 98367923874693874632559284619834677
```

in matra me firbec, da bi izvedel, koliko je `x ** n % m`. Vtipkati to v Python je slaba ideja, saj bo potenciral in dobil preveliko število, še preden se loti računanja ostanka. Pač pa lahko vpišemo

[13]:
```
pow(x, n, m)
```

[13]:
```
22872358624663969387800566354005582
```

Če sem na prvem predavanju omenil, da ima `pow` skrite supermoči: no, to. Izračunati zna ostanek po potenciranju, pri čemer se ne ustraši astronomsko velikih števil.

Preden ugotovimo, kako ta `pow` deluje (spoiler: tako kot skrita funkcija!), odgovorimo na prvo vprašanje: zakaj bi to potrebovali? Takšne potence - pri čemer je eksponent neko veliko praštevilo - so osnova več postopkov kriptografije. Tako deluje izmenjava ključev z RSA, z Diffie-Helmanovim postopkom… Vsakič, ko gre brskalnik na stran, ki podpira https (to je, dandanes: vsakič, ko gre na kakšno stran), se poračuna ena takšna potenca.

Zdaj pa k delovanju `pow`. Začeli bomo z (zeh, zeh) pretvarjanjem v dvojiški zapis. Imamo neko število, recimo

[14]:
```
int("100110", 2)
```

[14]:
```
38
```

38 (za katerega sem mimogrede izdal, da se po dvojiško zapiše `100110`), bi rad zapisal po dvojiško. Poznamo dril: ker je število sodo, zadnja števka 0. Nato ga delimo z `2` in dobimo 19 (oziroma `10011` - število v dvojiškem zapisu delimo z 2 tako, da odbijemo zadnjo števko). Ker je 19 liho, je zadnja števka 1. Spet delimo z 2 in dobimo 9 (oziroma, brez zadnje števka, `1001`). 9 je liho, torej je zadnja števka 1. Delimo z 2 in imamo 4 ali `100`. OK, sprogramirajmo to reč.

[15]:
```
def f(n):
    while n > 0:
        if n % 2 == 1:
            print(1)
        else:
            print(0)
        n //= 2
```

[16]:
```
f(38)
```

```
0
1
1
0
0
1
```

Zdaj pa mimogrede izpisujmo še dvojiške števke.

[17]:
```
def f(n):
    b = 1
    while n > 0:
        if n % 2 == 1:
            print(b, 1)
        else:
            print(b, 0)
        b *= 2
        n //= 2
```

[18]:
```
f(38)
```

```
1 0
2 1
4 1
8 0
16 0
32 1
```

Zdaj pa ne izpisujmo tistih ničel; pravzaprav izpisujmo samo števke, ki jih imamo pri enicah.

[19]:
```
def f(n):
    b = 1
    while n > 0:
        if n % 2 == 1:
            print(b)
        b *= 2
        n //= 2
```

[20]:
```
f(38)
```

```
2
4
32
```

Izpisalo se je 2, 4 in 32, kar je ravno prav, saj je $2 + 4 + 32 = 38$.

Pa recimo, da bi morali izračunati, koliko je 5^{38}.

[21]:
```
5 ** 38
```

[21]:
```
363797880709171295166015625
```

Počasi: $5^{38} = 5^{2+4+32} = 5^2 \times 5^4 \times 5^{32}$. To bi se dalo sprogramirati tako:

[22]:
```
def f(x, n):
    b = 1
    r = 1
    while n > 0:
        if n % 2 == 1:
            r *= x ** b
        b *= 2
        n //= 2
    return r
```

[23]:
```
f(5, 38)
```

[23]:
```
363797880709171295166015625
```

Da, številka je ista, torej funkcija deluje. Ampak lahko smo še pametnejši. Pravzaprav moramo biti: v funkciji še vedno uporabljamo potenciranje `x ** b`, kar je pravzaprav goljufija. Tega `x ** b` se moramo znebiti.

Poglejmo, kakšne potence uporabljamo: ko je `b` enak `1`, množimo (ali pa tudi ne) z x, nato je `b` enak `2` in morda množimo z x^2, nato je `b` enak `4` in morda množimo z x^4, nato z x^8, nato z x^{16}, x^32. Kako dobimo takšno zaporedje x-ov? S kvadriranjem! x^8 je ravno x^{4+4} oziroma $x^4 \times x^4$ oziroma ${x^4}^2$. Vsak naslednji x dobimo s kvadriranjem prejšnjega x-a.

Ukinemo torej `b`. Namesto z `x ** b` množimo kar z `x` in potem namesto, da bi podvojili `b`, pokvadriramo `x`.

[24]:
```
def f(x, n):
    r = 1
    while n > 0:
        if n % 2 == 1:
            r *= x
        x *= x
        n //= 2
    return r
```

[25]:
```
f(5, 38)
```

[25]:
```
363797880709171295166015625
```

In to je natančno funkcija iz uganke!

Kako hitra je ta funkcija? Še vedno imamo zanko. Vendar se je ona, naivna zanka prejšnje funkcija obrnila tolikokrat, kolikor je bil velik eksponent. Tale pa eksponent v vsakem koraku deli z 2 in

se ustavi, ko je eksponent enak 0. Ker deljenje z 2 v bistvu pomeni odbijanje zadnjega bita, se ta zanka obrne tolikokrat, kolikor bitov je velik eksponent. Če imamo 1024 bitni eksponent, se zanka obrne 1024-krat.

Samo še nekaj nam manjka: kako je s tistim ostankom? Vemo, da ne smemo najprej potencirati in šele nato računati ostanka.

To je pa preprosto: ker gre za sama množenja, lahko ostanke računamo kar sproti: ob vsakem množenju obdržimo le ostanek.

[26]:
```
def f(x, n, m):
    r = 1
    while n > 0:
        if n % 2 == 1:
            r = (r * x) % m
        x = (x * x) % m
        n //= 2
    return r
```

[27]:
```
f(x, n, m)
```

[27]:
```
22872358624663969387800566354005582
```