Návrh aplikace

IB111

Založeno na 9. kapitole knihy J. M. Zelle:
Python Programming: An Introduction to Computer Science

Simulace hry volejbal

- Chceme vytvořit simulátor hry volejbal
  - Ne grafický :-)
- Jde nám o počty vítězství 2 teamů, známe-li pravděpodobnosti výhry jejich podání.
- Například:
  - První team vyhrává svá podání s 60% úspěšností
  - Druhý team vyhrává svá podání s 50% úspěšností
- Úkolem je vytvoření programu, který bude simulovat několik her volejbalu
  - Používáme stará pravidla a existenci setů ignorujeme

Volejbal

- Začíná jeden team
  - Pokud získá své podání, dostane bod a opět podává
  - Pokud ztratí své podání, podává druhý team a bod nezískává nikdo
- Hra končí pokud nějaký team získá 15 bodů
  - Pokud je však rozdíl ve skóre pouze jednobodový, je třeba hrát dál dokud rozdíl není alespoň 2 body

Detailnější specifikace problému

- **Vstup:** Program se zeptá na potřebné údaje
  - Pravděpodobnost výhry podání teamu A
  - Pravděpodobnost výhry podání teamu B
  - Počet her, které budeme simulovat
- **Výstup:** Program vypíše výsledné statistiky
  - Počet simulovaných her
  - Počet her, které vyhrál team A
  - Počet her, které vyhrál team B

Náhoda

- Pro deterministický počítač není jednoduché vytvářet zcela náhodná čísla
- Pro simulace typicky používáme generátor pseudonáhodných čísel
  - Založené na „semínku“ (seed)
  - Vytváří posloupnost čísel s dobrými statistickými vlastnostmi
    - Vhodné pro simulace
- Python
  - Modul random
  - Pseudonáhodný generátor „Mersenne Twister “ inicializovaný aktuálním časem
    - Zcela nevhodné pro kryptografické účely

Generování náhodných čísel

- Python
  - random.randint(*a*, *b*)
    - Return a random integer *N* such that a <= N <= b.
  - random.random()
    - Return the next random floating point number in the range [0.0, 1.0).
- Výhra podání

```
if random() < prob:
    score = score + 1
```

kde prob je pravděpodobnost výhry hráče

Návrh shora dolů (top-down)

- Návrh začínáme na abstraktní (vysoké) úrovní
- Specifikujeme co principielně bude program dělat
- Uvedeme, jak se bude postupovat, aniž bychom museli rovnou zmínit všechny detaily jak to udělat
  - To upřesníme později
- V dalších krocích pak postupně upřesňujeme detaily, které jsme v předchozích krocích přeskočili
- Nakonec tak získáme kompletní program

Náš příklad se simulací volejbalu

- Základní algoritmus:

  Vypiš úvodní informace
  Získej vstupní údaje: probA, probB, n
  Simuluj n her teamů A a B za pomoci probA, probB
  Vypiš kolik her se hrálo, kolikrát vyhrál team A, B

- Pro každý krok dále musíme upřesnit vstupy a výstupy
  - parametry a návratové hodnoty funkcí
  - Interface / rozhraní

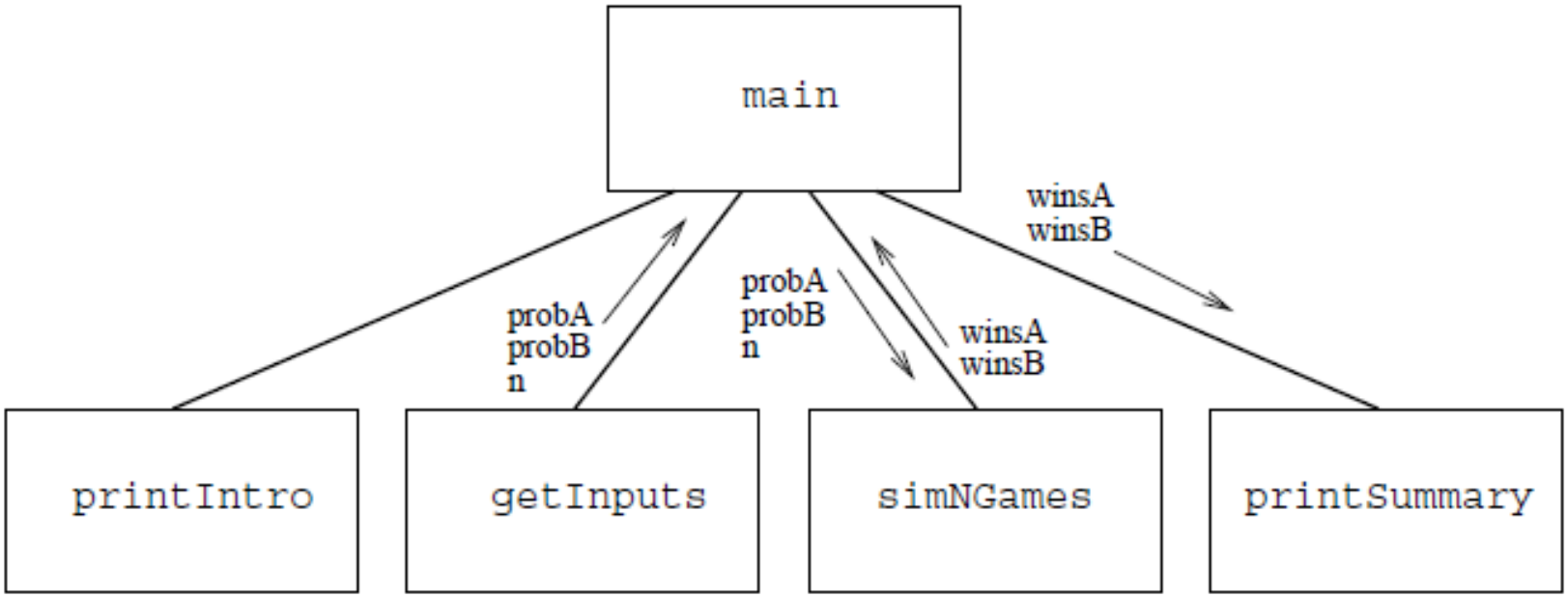
První návrh programu

- Program v pythonu:

```
def main():
    printInstructions()
    probA, probB, n = getInputs()
    winsA, winsB = simNGames(n, probA, probB)
    printSummary(winsA, winsB)
```

Grafický návrh

- Náš návrh lze zobrazit i graficky



Druhá úroveň návrhu

- Po navržení první (kořenové) úrovně návrhu pokračujeme druhou úrovní
- Specifikujeme chování funkcí použitých v návrhu první úrovně.
- Tj. specifikujeme funkce
  - printIntro
  - getInputs
  - simNGames
  - printSummary

Výpis úvodních informací

- Funkce v pythonu:

```
def printIntro():
    print "Tento program simuluje hru volejbal mezi 2"
    print 'teamy "A" a "B". Schopnosti teamu jsou dany'
    print "pravdepodobnostmi  (cislo mezi 0 a 1)  s jakymi"
    print "team vyhraje sve podani. Zacina vždy team A."
```

Získání vstupních dat

- Funkce v pythonu

```
def getInputs():
    a = input("S jakou pravdepodobnosti vyhraje
                team A sve podani")
    b = input("S jakou pravdepodobnosti vyhraje
                team B sve podani")
    n = input("Kolik her bude simulovano?")
    return a, b, n
```

Simulace n her

- Intuitivně bude simulace vypadat nějak takto:

  iniciálně počet výher winsA a winsB bude nula
  opakuj n-krát
      simuluj jednu hru
      jestliže vyhrál team A
          team A získává bod
      jinak
          team B získává bod

Simulace n her

- Přepíšeme do pythonu:

```
def simNGames(n, probA, probB):
    winsA = winsB = 0
    for i in range(n):
        scoreA, scoreB = simOneGame(probA, probB)
        if scoreA > scoreB:
            winsA = winsA + 1
        else:
            winsB = winsB + 1
    return winsA, winsB
```

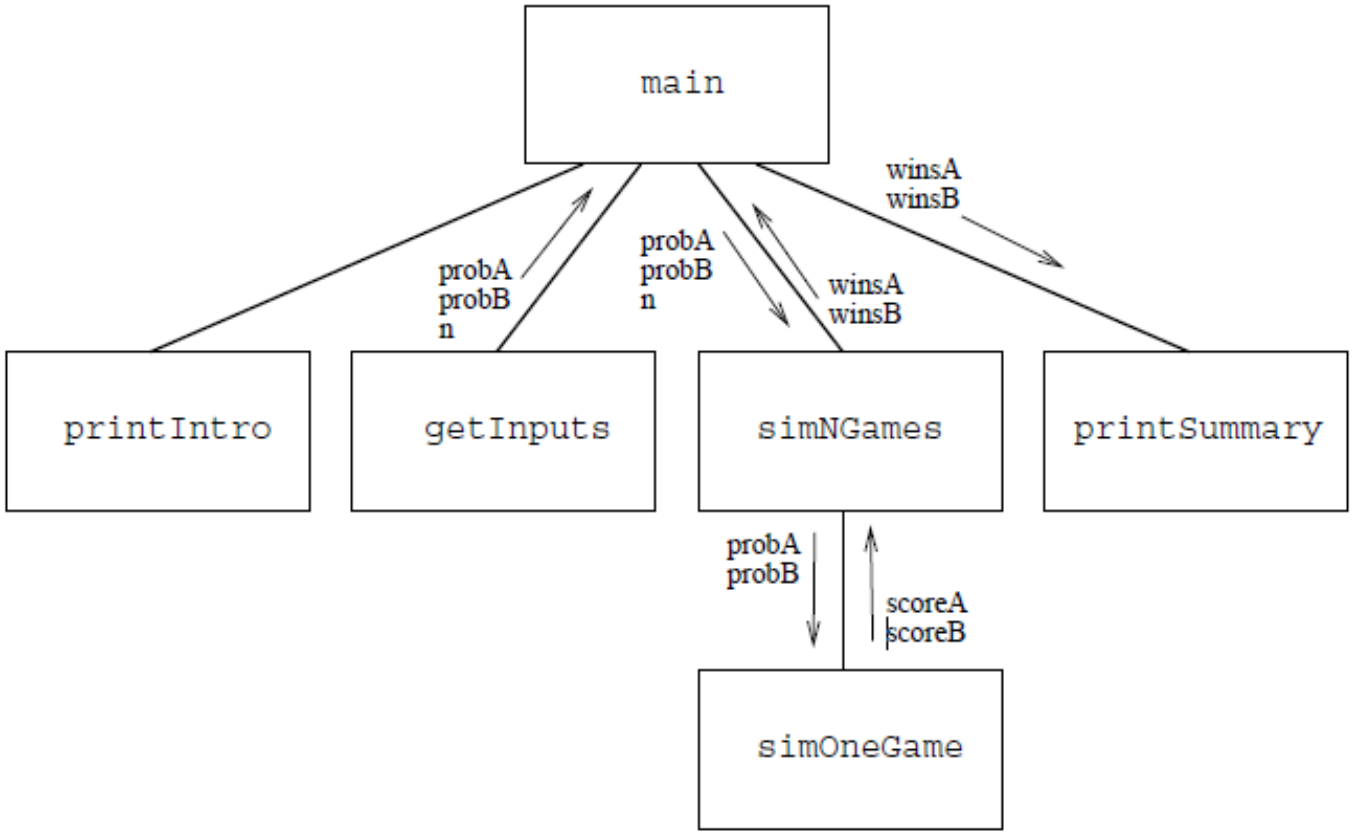
Výpis výsedku

- Zapíšeme v pythonu

```
def printSummary(winsA, winsB):
    n = winsA + winsB
    print "\nPocet simulovanych her:", n
    print "Pocet vyher A: %d (%0.1f%%)" % (winsA,
    float(winsA)/n*100)
    print "Pocet vyher B: %d (%0.1f%%)" % (winsB,
    float(winsB)/n*100)
```

Grafické znázornění

- Opět může aktuální specifikaci zobrazit graficky:



Třetí úroveň návrhu

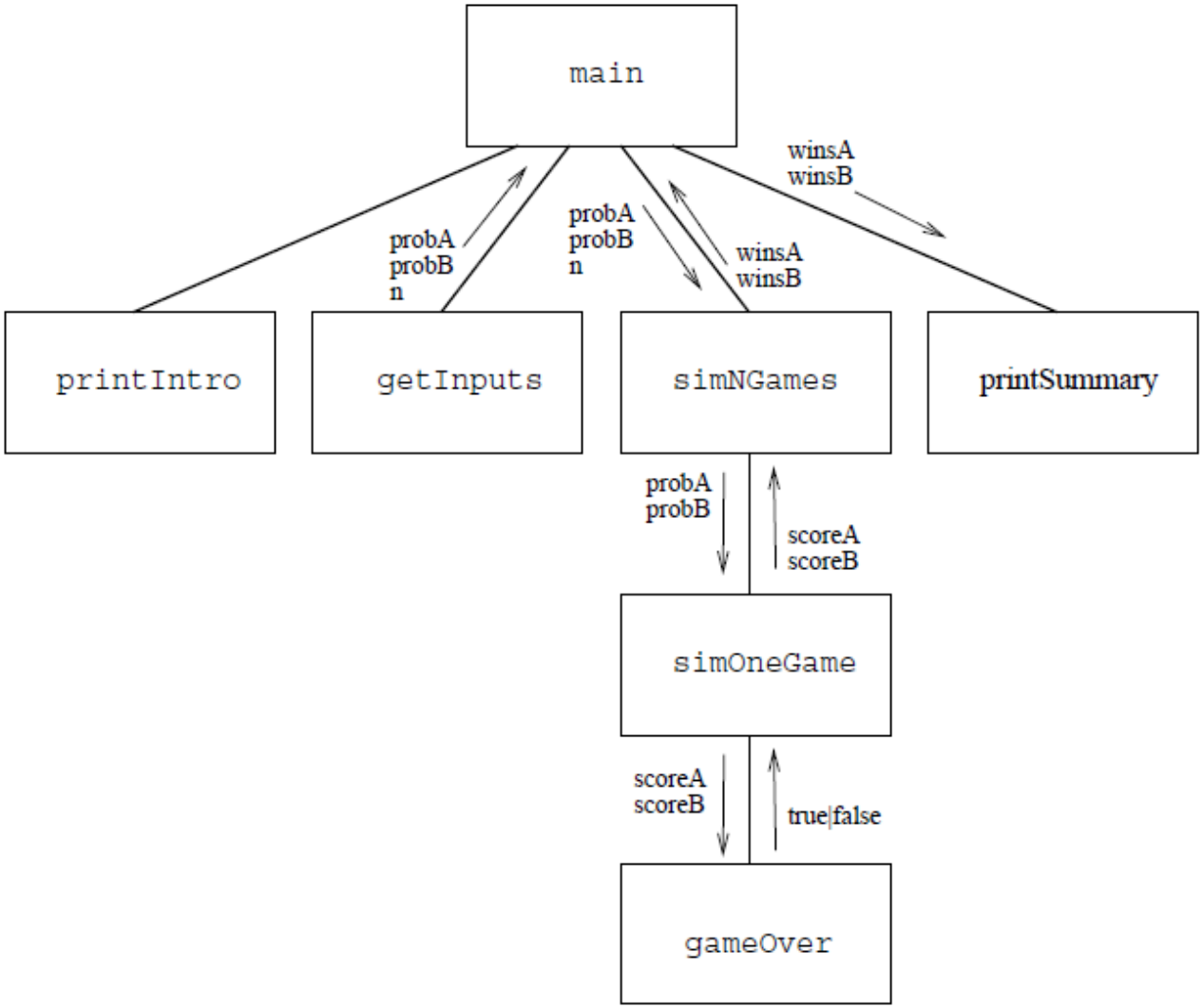
- Zbývá dospecifikovat funkci pro jednu hru
- Intuitivně bude funkce fungovat takto:

Inicializuje skóre obou teamů na nulu
Team A má podávat
Opakuj dokud není konec hry:
    simuluj jedno podání teamu, který podává
    aktualizuj skóre
Vrať skóre obou teamů

Implementace v pythonu

```
def simOneGame(probA, probB):
    scoreA = 0
    scoreB = 0
    serving = "A"
    while gameNotOver(scoreA, scoreB):
        if serving == "A":
            if random() < probA:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < probB:
                scoreB = scoreB + 1
            else:
                serving = "A"
    return scoreA, scoreB
```

Grafické zobrazení



Zbývá funkce pro zjištění konce hry

- Hra končí, pokud některý z teamů získá 15 bodů
  - A rozdíl skóre obou teamů je nejméně 2

```
def gameOver(a,b):
    return (a>=15 or b>=15) and (a>=b+2 or b>=a+2)
```

Jak jsme postupovali

1. Vyjádřili jsme algoritmus jako sérii menších problémů (bloků/funkcí)
2. Pro každý menší problém (blok/funkci) jsme vyvinuli rozhraní
3. Algoritmus jsem pak upřesnili tím, že jsme specifikovali, jak využijeme rozhraní s menšími problémy (bloky/funkcemi).
4. A obdobně jsme postupovali při návrhu jednotlivých menších problémů (bloků/funkcí) na nižších úrovních.

Testování

- Program implementujeme v pythonu
- Když ho poprvé spustíme je možné, že díky chybám nebude fungovat, tak jak si představujeme
  - Chybujeme všichni
  - Chyby jsou běžné
  - Bug
- Program budeme testovat po částech
  - Použijeme přístup „zdola nahoru“ (botton-up)
  - Nejprve ověříme funkčnost komponent na nižší úrovni a později komponent na vyšší úrovni

Testování funkce gameOver

- Vyzkoušíme na několika voláních s různými parametry:

```
>>> gameOver(15,15)
False
>>> gameOver(1,2)
False
>>> gameOver(1,15)
True
>>> gameOver(15,1)
True
>>> gameOver(16,15)
False
>>> gameOver(17,15)
True
>>>
```

- Pokud se zdá, že funkce funguje dobře, pokračujeme o úroveň výše.
- Pozn: Skutečná verifikace toho, že funkce dělá přesně to co má, je netriviální.

Jiné přístup k návrhu a vývoji SW

- Prototypování a spirálový vývoj
  - Prototyp – iniciální verze programu s omezenou funkčností
    - Vhodné pro ověření, zda principy, na kterých chceme SW založit fungují
  - Prototyp dále vylepšujeme, až získáme kompletní program s plnou funkčností
    - Inkrementální vývoj

Prototypování: úvodní verze

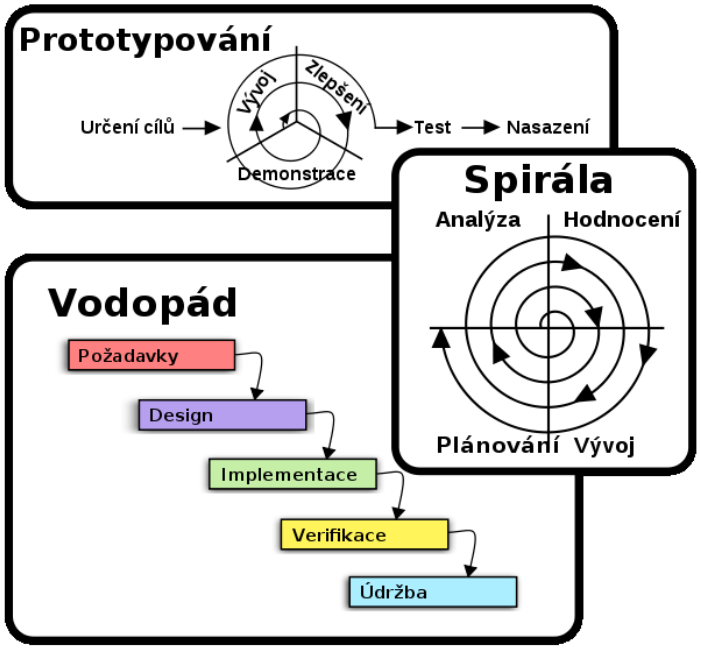
```
def simOneGame():
    scoreA = 0
    scoreB = 0
    serving = "A"
    for i in range(30):
        if serving == "A":
            if random() < .5:
                scoreA = scoreA + 1
            else:
                serving = "B"
        else:
            if random() < .5:
                scoreB = scoreB + 1
            else:
                serving = "A"
    print scoreA, scoreB
```

Prototypování: inkrementální vývoj

- Fáze:
  1. Iniciální prototyp: 30 podání s pravděpodobností vyhrání podání 50 % a 50 %. Pomocný výpis po každém podání.
  2. Přidáme parametry pro pravděpodobnosti výher podání.
  3. Přidáme funkci sledující konec hry při získání 15 bodů jedním hráčem. Tím máme funkční simulaci jedné hry.
  4. Rozšíříme na opakování několika her. Výstupem bude počet výher jednotlivých teamů.
  5. Vývoj plně funkčního programu. Dořešení získání vstupu od uživatele a vhodného formátování výstupu.

Vývoj softwaru

- Vývoj softwaru je umění
- Již dlouhou dobu se jí zabývá věda nazývaná „Softwarové inženýrství“ („Software engineering“)
- S dalšími aspekty vývoje SW se setkáte v dalších předmětech na FI
  - například PB007 Softwarové inženýrství I



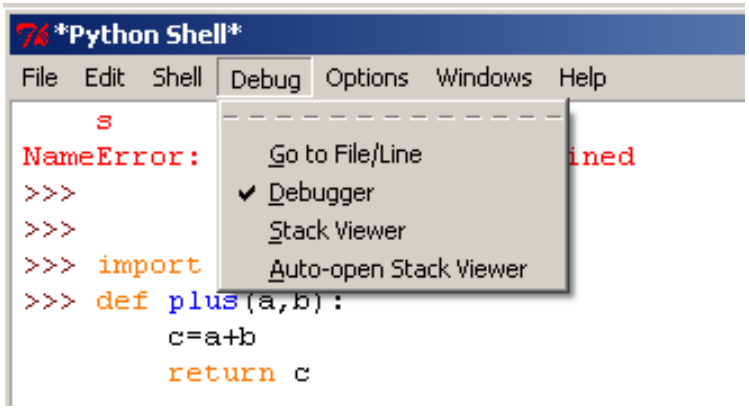
Ladění programů v pythonu

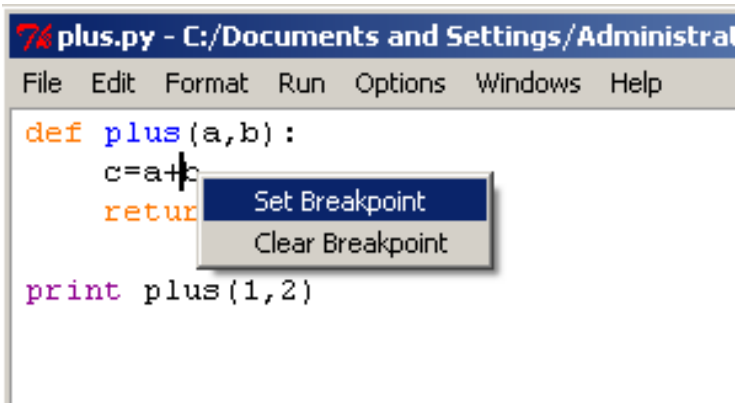
- Modul pdb
- Spustíme ladění vykonávání určitého kódu
- Nebo program zastavíme na určitém místě (breakpoint) pomocí pdb.set_trace()
- Potom můžeme
  - Pokračovat (c)
  - Krokovat (s)
  - Prohlížet proměnné (p)
  - …

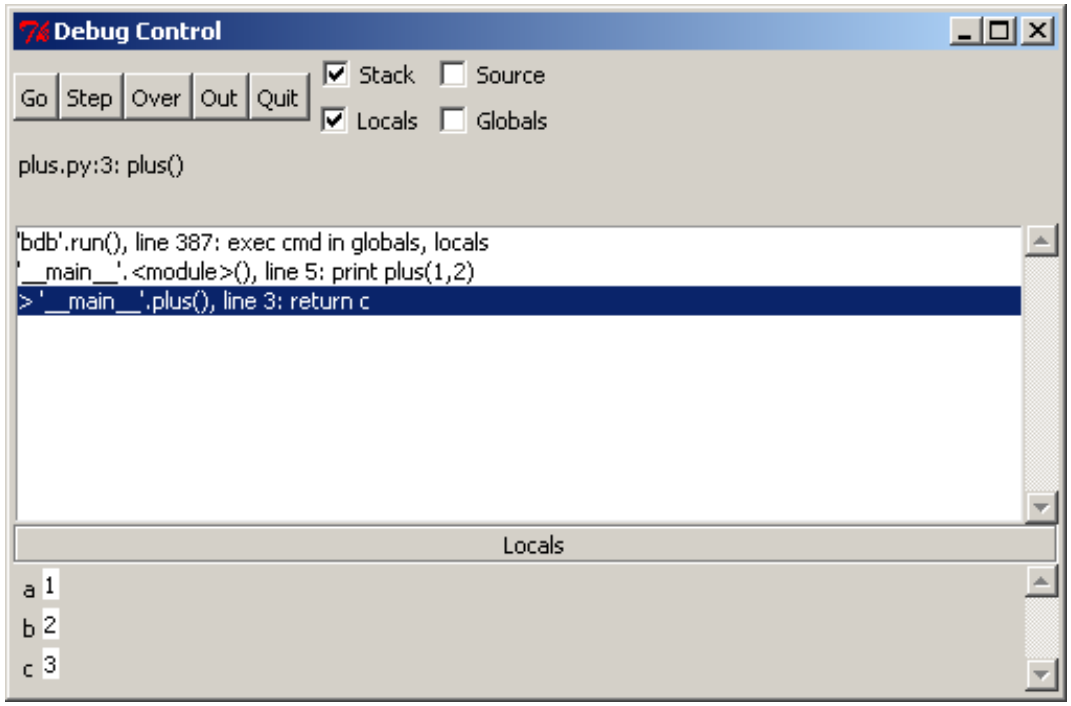
```
>>> import pdb
>>> def plus(a,b):
        c=a+b
        return c

>>> pdb.run('plus(1,2)')
> <string>(1)<module>()->None
(Pdb) s
--Call--
> <pyshell#38>(1)plus()
(Pdb) s
> <pyshell#38>(2)plus()
(Pdb) s
> <pyshell#38>(3)plus()
(Pdb) p c
3
(Pdb) s
--Return--
> <pyshell#38>(3)plus()->3
(Pdb) s
--Return--
> <string>(1)<module>()->None
(Pdb) s
>>>
```

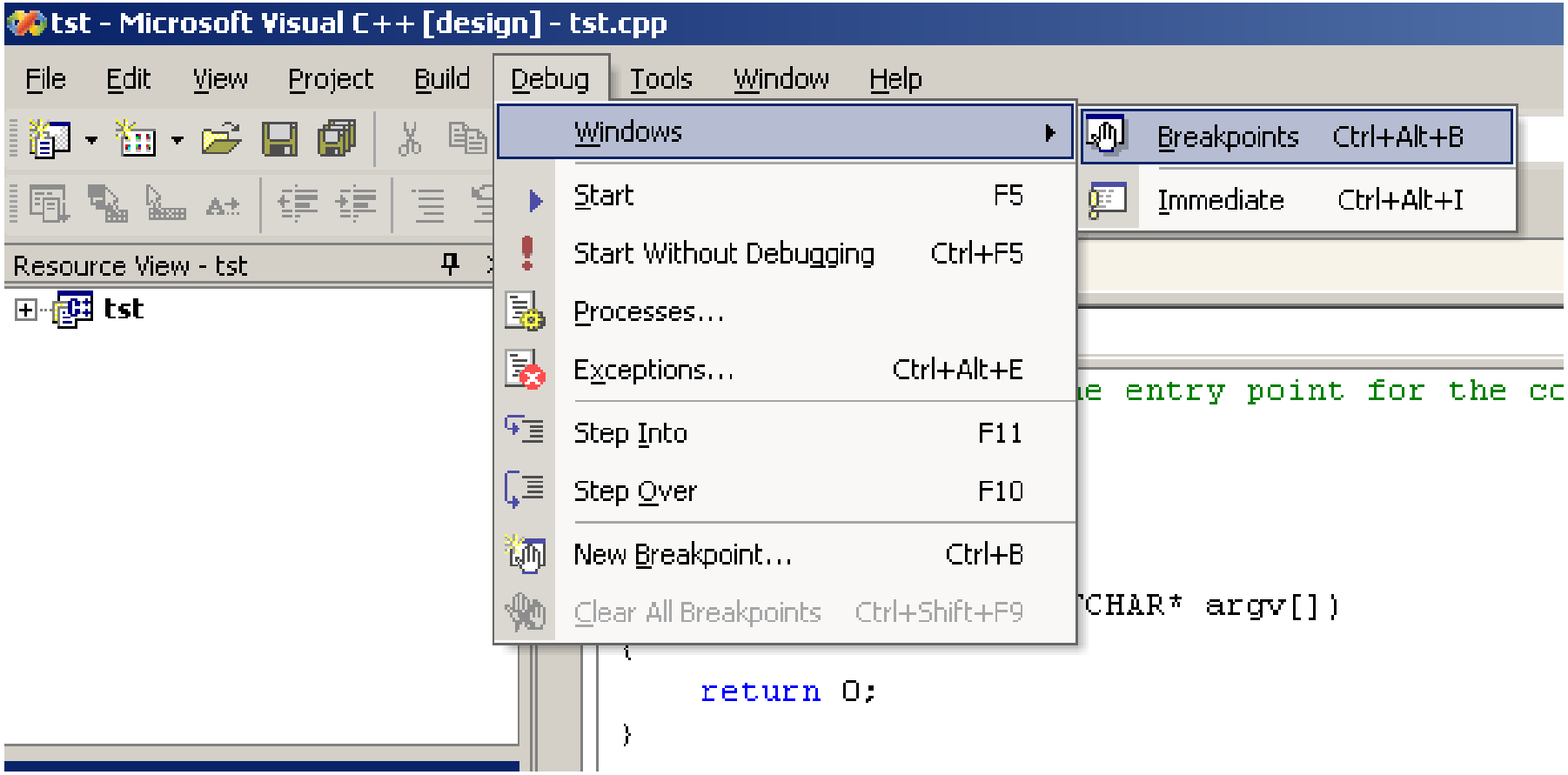
Ladění programů v Pythonu v IDLE

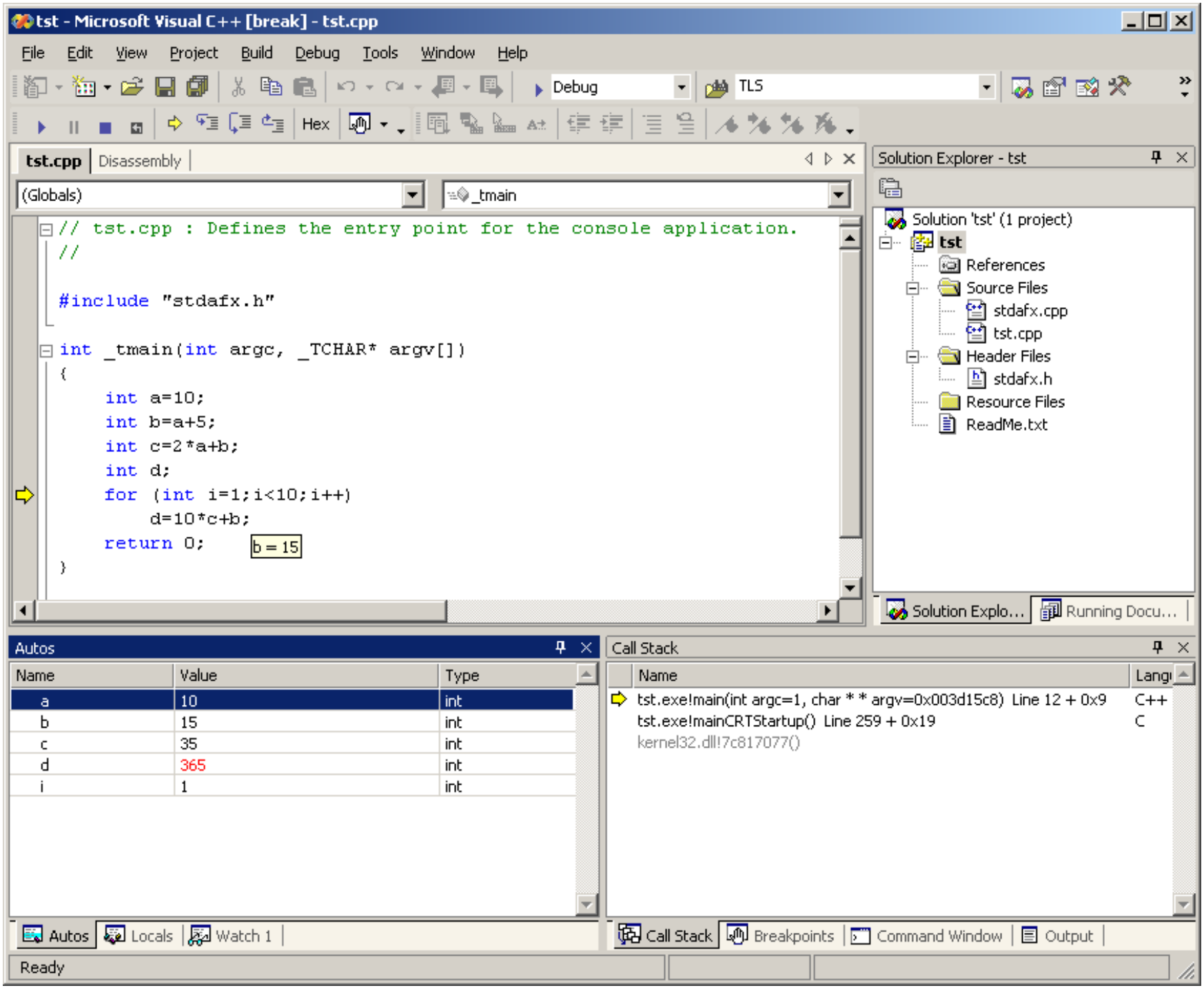






Ladění programů v C: Visual Studio C++




tst - Microsoft Visual C++ [break] - tst.cpp
File Edit View Project Build Debug Tools Window Help
Debug
TLS
Hex
tst.cpp
Disassembly
(Globals)
_tmain
// tst.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
int a=10;
int b=a+5;
int c=2*a+b;
int d;
for (int i=1;i<10;i++)
d=10*c+b;
return 0;
b = 15
}
Solution Explorer - tst
Solution 'tst' (1 project)
tst
References
Source Files
stdafx.cpp
tst.cpp
Header Files
stdafx.h
Resource Files
ReadMe.txt
Solution Explo...
Running Docu...
Autos
Name
Value
Type
a
10
int
b
15
int
c
35
int
d
365
int
i
1
int
Autos
Locals
Watch 1
Call Stack
Name
Lang
tst.exe!main(int argc=1, char * * argv=0x003d15c8) Line 12 + 0x9
C++
tst.exe!mainCRTStartup() Line 259 + 0x19
C
kernel32.dll!7c817077()
Call Stack
Breakpoints
Command Window
Output
Ready

Dokumentování programů

- Poznámky/komentáře
  - Usnadňují pochopení jednotlivých konstrukcí
  - Píšeme pro sebe i ostatní
    - Užitečné pro pozdější modifikace programu
      - Prováděné námi i jinými
- Dokumentace rozhraní
  - API
  - Ty funkce/moduly/knihovny, které budou používány ostatními
- Názvy modulů, funkcí, proměnných…

Dokumentace rozhraní

- Funkce
  - Jméno funkce
  - Vstupní parametry
    - Povinné
    - Nepovinné
    - Defaultní hodnoty
  - Návratová hodnota
  - Funkčnost

Příklad z dokumentace Pythonu

`random.`**`randint`**(*a*, *b*)

Return a random integer *N* such that `a <= N <= b`.

`string.`**`capitalize`**(*word*)

Return a copy of *word* with only its first character capitalized.

`string.`**`expandtabs`**(*s*[, *tabsize*])

Expand tabs in a string replacing them by one or more spaces, depending on the current column and the given tab size. The column number is reset to zero after each newline occurring in the string. This doesn't understand other non-printing characters or escape sequences. The tab size defaults to 8.

Příklad z dokumentace Windows API

Platform SDK: Windows System Information

GetFileTime

The **GetFileTime** function retrieves the date and time that a file was created, last accessed, and last modified.

```
BOOL GetFileTime(
  HANDLE hFile,
  LPFILETIME lpCreationTime,
  LPFILETIME lpLastAccessTime,
  LPFILETIME lpLastWriteTime
);
```

Parameters

hFile
[in] Handle to the files for which to get dates and times. The file handle must have been created with the GENERIC_READ access right. For more information, see File Security and Access Rights.

lpCreationTime
[out] Pointer to a **FILETIME** structure to receive the date and time the file was created. This parameter can be NULL if the application does not require this information.

lpLastAccessTime
[out] Pointer to a **FILETIME** structure to receive the date and time the file was last accessed. The last access time includes the last time the file was written to, read from, or, in the case of executable files, run. This parameter can be NULL if the application does not require this information.

lpLastWriteTime
[out] Pointer to a **FILETIME** structure to receive the date and time the file was last written to. This parameter can be NULL if the application does not require this information.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Dokumentujeme v Pythonu

- Dokumentační řetězce (docstring)
- První řetězec třídy, modulu, funkce (metody)
- Všechny „veřejné“ části by měly být dokumentovány

```
def soucet(a,b):
    """ Funkce vraci soucet dvou argumentu. """
    return a+b
```

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0: return complex_zero
    ...
```

Jak správně dokumentovat?

4.7.6. Documentation Strings

There are emerging conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace "equivalent" to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Příklad využití

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

```
>>> print ord.__doc__
ord(c) -> integer

Return the integer ordinal of a one-character string.
>>>
```

Styl

- Styl psaní programů
  - Styl programování, coding style
- Stejný program lze zapsat různým způsobem
  - Pěkně vs. škaredě (nevhodně)

Doporučení pro Python

- Odsazení: 4 mezery
- Délka řádku: pod 80 znaků
- Prázdné řádky mezi funkcemi a třídami
- Komentáře na separátních řádcích
- Mezery mezi operátory

```
>>> a = f(1, 2) + g(3, 4)
```

  - Ale žádné mezery u závorek při volání funkcí
- Rozumné a konzistentní názvy funkcí
  - dlouhy_nazev_funkce
  - DlouhyNazevFunkce

Příklady

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )
```

```
Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

```
Yes: spam(1)
No:  spam (1)
```

Yes:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Yes:

```
x = 1
y = 2
long_variable = 3
```

No:

```
x             = 1
y             = 2
long_variable = 3
```

```
Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x
```

Další příklady

Yes:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

Definitely not:

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```