SmSL Dokumentace

Smajlík

Obsah

1. Úvod

1.1 Co je SmSL
SmSL je interpretovaný programovací jazyk napsaný v Pythonu. Nevím co jiného sem napsat :D

2. Základy jazyka SmSL

Tato část popisuje, jak psát programy v jazyce SmSL.
Všechny programy v jazyce SmSL musí začínat řádkem #include ”smsl.h”. Tento řádek totiž načte soubor s definicemi, podle kterého se řídí interpret při provádění vašeho kódu. Více informací o těchto souborech naleznete v části ”Soubory definic”.

2.1. If, else if, else

If, else if, else píšeme v SmSL takto:

```
if <podmínka> (
    # Kód, který se má provést při splnění podmínky.
)
else if <další podmínka> (
    # Kód, který se má provést při splnění podmínky.
)
else (
    # Tento kód se provede, pokud není splněna žádná podmínka.
)
```

2.2 Smyčky for a while

2.2.1 Smyčka for
Smyčka for se v SmSL zapisuje takto:
```
for <proměnná> in <objekt> nebo range[<číslo>]  (
    # Tento kód se bude opakovat.
)
```
<proměnná> je název proměnné, do které se při každém opakování uloží prvek seznamu/slovníku,
<objekt> je řetezec/seznam/slovník, ke kterému přistupujeme.
<číslo> je počet opkování kódu. Pokudje potřeba, tak <proměnných> můžeme mít samozřejmě víc.

2.2.2 Smyčka while
Smyčka while se v SmSL zapisuje takto:
```
while <podmínka> (
    # Tento kód se bude opakovat dokud platí podmínka
)
```

2.3 Try, catch

try/catch se v SmSL zapisuje takto:
```
try (
    # Kód sem
)
catch <chyba> (
    # Kód zde se provede, pokud uvnitř bloku try nastane <chyba>.
```

```
)
```

<chyba> je chyba, která musí nastat v try bloku, aby se provedl kód v bloku except.

2.4 Některé důležité funkce

V této části jsou parametry funkcí (mimo jejich výpisů) ohraničeny znaky < a >.

```
write[text, end="\n", file=sys.stdout]
```

Funkce write vypíše <text> do terminálu (nebo do souboru). Na konec textu se připojí <end>. Parametr <file> určuje, kam se má text vypsat, sys.stdout označuje standardní výstup.

```
num[string]
```

Tato funkce není tak úplně funkcí. Je to ve sktuečnosti třida (o těch se dozvíte později), ale myslím si, že si zaslouží být zde uvedena. Převádí <string> na číslo. <string> může obsahovat pouze číslice. V případě že <string> obsahuje něco jiného než číslice vyhodí chybu.

```
readf[filename, binary=False]
```

Funkce readf načte obsah souboru s názvem <filename>. Parametr <binary> určuje, jestli se má číst v binárním režimu. To je užitečné pro čtení např. Obrázků a jiných souborů, které by jinak tato funkce nemohla přečíst.
Všechny ostatní funkce jsou stejné jako v jazyce Python. Pro informace o nich navštivte dokumentaci Pythonu.

2.5 Funkce

Vedle používání funkcí uvedeých výše můžeme v jazyce SmSL také vytvářet vlastní funkce. To se provádí takto:

```
func jmeno_funkce[arg1, arg2, arg3..., kwarg1=val1, kwarg2=val2,
kwarg3=val3...] (
    # Kód funkce sem
)
```

<jmeno funkce> je jméno funkce, <arg1, arg2, arg3...> jsou argumenty funkce. <kwarg1, kwarg2, kwarg3>jsou tzv. klíčové argumenty. Tak říkáme argumentům, kterým je již v definici funkce přiřazena hodnota a pokud tuto hodnotu nepotřebujeme měnit, tak je nemusíme zadávat. Můžeme také vytvářet funkce, které žádné argumenty nepřijímají. V takovém případě napíšeme za jméno funkce prázdné hranaté závorky. Můžeme dokonce vytvářet i funkce beze jména.
To se dělá takto:

```
[func] a ( write[a] )
```

Takové funkce využijeme např. pokud potřebujeme jiné funkci zadat argumenty aniž bychom ji spouštěli. Ve funkcích můžeme používat některé příkazy, které mimo funkce používat nemůžeme. Například příkaz return, který nám umožňuje z funkce vrátit hodnotu nějaké proměnné. Používá se takto:

```
func jmeno_funkce[arg1] (
    # Kód funkce zde
    return arg1
)
```

Protože příkaz return zastaví provádění funkce, tak pokud chceme vrátit více hodnot, musíme použít příkaz yield. Pokud je příkaz return nebo yield použit mimo funkci, vyhodí chybu. Pokud použijeme ve funkci příkaz yield, tak musíme pro získání hodnot z funkce použít smyčku for.

Příklad:

```
func vypis_prvku[seznam] (
    for i in seznam (
        yield i
    )
)

for i in vypis_prvku[{1, "abc", "SmSL"}] (
    write[i]
)
```

Výstup programu:
1
abc
SmSL

2.6 Třídy

Třídy v SmSL se vytvářejí takto:

```
class JmenoTridy (
    func init[this, arg1] (
        var this.arg1 = new arg1
    )
    func vrat_arg1[this] (
        return this.arg1
    )
)
```

Všechny funkce ve třídách musí přjímat argument this a to vždy jako první argument (kromě funkcí nad kterými je příkaz `@staticmethod`). Proměnné
vytvořené pomocí this lze používat v celé třídě. Při spouštění jakékoli funkce ve třídě agument this vynecháváme. Poté co pomocí této třídy vytvoříme objekt, můžeme k takto vytvořeným proměnným přistupovat i mimo třídu. Funkce init je spuštěna automaticky po vytvoření objektu.
Příklad:

```
class JmenoTridy (
    func init[this, arg1] (
        this.arg1 = arg1
    )
    func vrat_arg1[this] (
        return this.arg1
    )
)

var objekt = new JmenoTridy["text"]
write[objekt.vrat_arg1[]]
write[objekt.arg1]
var objekt.arg1 = "abcd"
write[objekt.vrat_arg1[]]
write[objekt.arg1]
```

Výstup programu:
text
text
abcd
abcd
Další informace o třídách zde: https://naucse.python.cz/course/pyladies/beginners/class/ Článek na adrese výše sice pojednává o třídách v Pythonu, třídy v SmSL však fungují prakticky stejně.

2.7 Deklarace proměnných

Deklalrace čili vytvoření proměnné se v SmSL provádí takto:

```
var nazev_promenne = new "SmSL je nejlepší!"
```

V současné verzi SmSL je možné slova var a new vynechat. Můžeme tedy psát pouze:

```
nazev_promenne = „SmSL je nejlepší“
```

Obsahem proměnné může být prakticky cokoliv. Může to být text, číslo, seznam, slovník, true/false, funkce, třída atd.

2.8 Seznamy

Seznamy nám umožňují uložit si do jedné proměnné více hodnot. Vytváříme je zapsáním všech prvků seznamu oddělených čárkami do složených závorek K jednotlivým hodnotám (říkáme jim prvky seznamu) pak přistupujeme pomocí jejich čísla. První hodnota má číslo 0, druhá má číslo 1, třetí má číslo 2 atd. Obecně platí, že číslo prvku v seznamu získáme tak, že od jeho pořadí v seznamu odečteme 1. Např. máme seznam {1, ”abcd”, true}. Chceme-li zjistit číslo prvku ”abcd”, tak vidíme, že prvek je v seznamu druhý, odečteme 1 a dostaneme číslo 1. Seznamy v SmSL vytváříme a pracujeme s nimi takto:

```
# Vytvoření seznamu
var muj_seznam = new {1, "abcd", true}
# Výpis 2. prvku seznamu
write[muj_seznam{1}]
# Změna 3. prvku seznamu
var muj_seznam{2} = false
# Přidání prvku do seznamu
muj_seznam.append["SmSL"]
# Výpis všech prvků v seznamu
for i in muj_seznam (
    write[i]
)
```

Výstup programu:
abcd
1
abcd
false

Jak je vidět v příkladu výše, prvky do seznamu přidáváme pomocí funkce append. Ta přijímá jako argument hodnotu, kterou chceme přidat do seznamu. Pokud chceme ze slovníku vytvořit text, napíšeme <oddělovač>.join[<seznam>], kde oddělovač je znak, který má být vložen mezi jednotlivé prvky seznamu. Příklad:

```
mylist = {„a“, „b“}
write[„/“.join[mylist]]
```

Výstup programu:
a/b

Pokud obsah slovníku uzavřeme do
hranatých závorek, jeho obsah nebude možné měnit. Stejného efektu dosáhneme, pokud závorky vynecháme úplně a zapíšeme pouze prvky oddělené čárkami. Takovým seznamům říkáme tuple, česky n-tice.

2.9 Slovníky

Slovníky jsou něco jako seznamy, ale jeho prvky mají místo čísel názvy. Název oddělujeme od prvku dvojtečkou. V SmSL je vytváříme a pracujeme s nimi takto:

```
# Vytvoření slovníku
var muj_slovnik = new {| "jednicka" : 1, "abeceda" :
"abcdefghijklmnopqrstuvwxyz",
"true" : true
|}
# Výpis 2. prvku slovníku
write[muj_slovnik{"abeceda"}]
# Změna 3. prvku slovníku
var muj_slovnik{"true"} = false
# Přejmenování prvku
var muj_slovnik{"false"} = muj_slovnik{"true"}
rem muj_slovnik{"true"}
# Výpis všech prvků a jejich názvů
for i, j in muj_slovnik.items[] (
    write[i]
    write[j]
)
```

Výstup programu:
abcdefghijklmnopqrstuvwxyz
jednicka
1
abeceda
abcdefghijklmnopqrstuvwxyz
false
false

Všimněte si, že slovník vytváříme pomocí {| a |}, ale k jeho prvkům přistupujeme pomocí složených závorek bez znaku |. Všimněte si také, že k vypsání názvů a samotných prvků jsem použil funkci items[] a dvě proměnné ve for cyklu. Pokud bych napsal pouze for i in muj slovnik, musel bych prvky vypisovat jako muj slovnik[i].

2.10 Práce se soubory

Tato podkapitola obsahuje základy práce se soubory v SmSL.
Soubory v SmSL otevíráme takto:

```
var soubor = new open["název_souboru", "r"]
```

Pokud chceme otevřít soubor pro zápis, místo ”r”napíšeme ”w”. Otevřením souboru pro zápis se jeho obsah vymaže. Chceme-li zapisovat do souboru be jeho vymazání, napíšeme ”a”místo ”w”. Z otevřeného souboru čteme funkcí read nebo readlist. Funkce read vrací text, funkce readlist vrací seznam řádků (každý prvek je jeden řádek souboru). Do souboru zapisujeme pomocí funkce write. Této funkci musíme samozřejmě zadat, co má do souboru zapsat.

3. Soubory definic a definice

Všechny programy v jazyce SmSL musí začínat řádkem #include ”smsl.h”. Tento řádek totiž načte soubor s definicemi, podle kterého se řídí interpret při provádění vašeho kódu. Soubory definic jsou soubory, které interpretu jazyka SmSL říkají, jak má co přeložit. Programy v jazyce SmSL jsou totiž před vlastním spuštěním přeloženy do jazyka Python.
Soubory definic vypadají takto:

#define něco něco+jiného
Tento řádek se několikrát opakuje, pokaždé s jinými hodnotami (místo něco a něco-jiného). Tento příkaz můžeme využívat také přímo v kódu. Pokud potřebujeme v kódu změnit třeba ++ na += 1, použijeme příkaz #define \+=+1 \+\+. Před znakem plus musíme uvést zpětné lomítko, protože plus bez něj v tomto příkazu zastupuje mezeru. V tomto příkazu mezera odděluje od sebe text, který se má nahradit, a text kterým se má ten druhý nahradit (v opačném pořadí), proto nemůžeme pro nahrazení textu obsahujícího mezeru použít právě mezeru. Opakem příkazu #define je příkaz #undef. Ten zruší zadané pravidlo pro nahrazení, např. #undef \+= nám zruší pravidlo nahrazující nějaký text znaky += pokud takové pravidlo existuje. V opačném případě vyhodí chybu.