-- DATA BLOCKS --

[Author: Mikhail V., 2018]

-- 1 -- Definition and basic examples

:::

"Data block" is a syntactical construct suggested for defining data trees and simple data entries like tuples, lists, etc. Main aim is to remove punctuation redundancy and improve formatting consistence in multi-line data blocks. As a result, this should improve the readability of data and lower the user input efforts within such data blocks.

Main syntax principle:
- nesting (and parsing) by indentation, similar to all Python blocks;
- start of the suite: "///" -- triple slash;
- first token after "///" defines the type of structure, e.g. "/// t" for tuples;
- whitespace separated elements;
- expressions are put in parentheses

All examples provided in order: "new syntax" vs. "current Python syntax"
Note that best results can be achieved with syntax highlighting, so by the judgement it should be understood that e.g. type descriptors in real situation should be highlighted with different color than items.

:::

~ example 1.1 -- tuple

```
/// t   1
/// t   1  2  3
/// t   (x + y)  (sin(x))
```

Python form:

```
(1,)
(1, 2, 3)
(x + y,  sin(x))
```

~ example 1.2 -- assignment

```
data = /// t   1  2  3
data = /// t
    1  2  3
data =\
/// t
    1  2  3
```

```
data = (1, 2, 3)
data = (
    1, 2, 3 )
data = (
    1, 2, 3
)
```

~ example 1.3 -- nesting

```
data =\
/// t
    /// .
        /// .   1   2   3
        /// .   123   332   543
    /// .    "hello"   "foo bar"
```

```
data =\
(
    (
        (1, 2, 3)  ,
        (123, 332, 543)    ) ,
    ("hello", "foo bar")   )
```

> *Note: the dot after /// means inheritance of type from the previous suite <

-- 2 -- Basic types of data blocks

:::
Main types of blocks are:
“t” -- tuple
“L” -- list (uppercase is chosen to avoid visual ambiguity "1lI")
“d” -- dictionary
Dictionaries are parsed by simple rule: odd elements are keys,
even elements are values.
:::

~ example 2.1 -- mixed-type block

```
data =\
/// t
    /// .
        /// L   1   2   3
        /// L   123   332   543
    /// d   "hello"   "foo bar"
```

```
data =\
(
    (
        [1, 2, 3]  ,
        [123, 332, 543]   ) ,
    {"hello": "foo bar"}   )
```

~ example 2.2 -- nested dictionary

```
data =\
/// d
    "apples"
    /// .      "type"       "fruit"
               "color"      "green"
               "amount"  5
    "carrots"
    /// .      "type"       "vegetable"
               "color"      "orange"
               "amount"  20

# vs
data =\
{
    "apples": {
        "type":          "fruit" ,
        "color":     "green" ,
        "amount": 5  },
    "carrots": {
        "type":      "vegetable" ,
        "color":         "orange" ,
        "amount": 20  }
}
```

::: As seen in last example, the Python equivalent becomes very “noisy” due to excessive punctuation. It requires significant effort for user input and also may lead to ‘misreading’ of data :::

-- 3 -- Slicing, getting items, stacking

~ example 3.0 -- getting items / slicing

```
data = /// t[0:2]     1   2   3
data =\
 /// t[0]
     /// t[1]     1   2   3
     /// t         4   5   6
#vs
data = (1, 2, 3)[0:2]
data = ((1, 2, 3)[1], (4, 5, 6))[0]
```

:::
Stacking: further contraction can be achieved by additional type descriptors for stacking of leaf nodes (external nodes).
i.e.: 2-level stacking:
  /// t*t ...
3-level stacking:
  /// t*t*t ...
:::

~ example 3.1 -- two-level stacking (e.g. tuple of tuples)

```
data = /// t*t     1   2  ;   3   4  ;   5   6  ;   7   8
data = /// t*t
     "foo"    "bar"
     "apple"    "orange"    "plum"
#vs
data = (     (1, 2),  (3, 4),  (5, 6),  (7, 8) )
data = (
     ( "foo", "bar" ),
     ( "apple", "orange", "plum" )  )
```

>Note: for two-level stacking: semicolon separates external branches;
  newline has same function as semicolon.<

~ example 3.2 -- three-level stacking (tuple of tuples of tuples)

```
data = /// t*t*t
     1  2  ;  3  4
     5  6  ;  7  8
#vs
data = (
     ((1, 2),  (4, 4) ),
     ((5, 6),  (7, 8) ),    )
```

>Note: for 3-level stacking, newline separates sub-external branches.<

~ example 3.3 -- binary tree (four levels of nesting)

```
tree4 = \
/// t
    /// .
        /// t*t*t
            11  22  ;   11  22
            11  22  ;   11  22
        /// t*t*t
            11  22  ;   11  22
            11  22  ;   11  22
    /// .
        /// t*t*t
            11  22  ;   11  22
            11  22  ;   11  22
        /// t*t*t
            11  22  ;   11  22
            11  22  ;   11  22
#
#vs
tree4 = \
    (
        (
            (
                ((11, 22), (11, 22)),
                ((11, 22), (11, 22))
            ),
            (
                ((11, 22), (11, 22)),
                ((11, 22), (11, 22))
            )
        ) ,
        (
            (
                ((11, 22), (11, 22)),
                ((11, 22), (11, 22))
            ),
            (
                ((11, 22), (11, 22)),
                ((11, 22), (11, 22))
            )
        )
    )
```

-- 4 -- Additional types

-- 4.1 -- Strings

:::
In addition to common Python tokens, additional
“strings” presentation form is suggested. String presentation
is aimed at scenarios where all or most of the elements are strings.
(for this time the type “strings” is marked with a “s” in the descriptor)
Rules:
- parentheses enclose strings containing spaces.
- back-tick leaves the token as is
:::

~ example 4.1 -- strings only

```
/// ts   apples   bananas   plums   (pink grapefruit)
#vs
("apples",  "bananas",  "plums",  "pink grapefruit")
```

~ example 4.2 -- mixed with non-strings

```
/// ds    foo  `12      sum  `(x + y)      par   \(\)      es  ()
#vs
("foo":  12,  "sum":  x + y,  "par": "()",  "es":  "")
```

> Note: it is suggested to borrow f-string functionalty for these strings by default.
Or maybe make new similar syntax, for example: <

```
/// ts   {92 93 94}abc   key:{t}value{w:}{r n}
#vs
(f"{92:c}{93:c}{94:c}abc", "key:\tvalue{w}\r\n" )
```

-- 4.2 -- Multi-line string

:::
This is a controversial suggestion, since this kind of object
requires a dedicated type and it is not a collection, but rather
a single variable. Also syntax-highlighting will not work properly
by default. Motivation is to be able to define indentation-aware
multi-line strings (see “Notes” in the end of document).
:::

```
/// MS
        A string that spans over several lines and can be
        defined in any block without need to use "dedent".
```

--- SUMMARY ---

:::
Main justification for the proposed syntax is relatively wide range of cases where it is applicable. Most of mentioned examples show improvements in readability and potential for reduction of maintenance time (input and editing of data).
The benefits can be bigger in projects with a lot of data definitions. Being part of syntax, it will enable a user to directly import resources into projects or define them directly in relevant parts of code, which is very common for testing.

Points on visual appeal and readability:

- Reduction of redundant punctuation.
- Consistent formatting similar to the rest of Python blocks.
- Explicit type descriptors eliminates the visual similarity found in current collection constructs: []{}(), which all use brackets and sometimes for different purposes, e.g. index slicing.
- Whitespace separated elements are widely used in all kinds of layouts, educational materials, etc. so it reduces entrance barrier for new learners.
- Aesthetics is an important factor in language learning and usage

Points on input simplicity:

- Reduction of redundant punctuation.
- No need to go back and forth to edit the brackets: just edit the type descriptor to change the type of structure.
- Forcing of consistent formatting reduces efforts in long-term maintenance.
- Simpler copy-pasting, commenting-out.
- Getting an item directly in the structure is useful for testing

Main problems:

- Complexity of parser, clash with current parts, e.g. line continuation.
- New syntax will create different source styles (adoption problem).
- Some of proposal parts (type descriptors, “strings” type) require modifications to text editors’ syntax highlighting to work properly.
- Many opened questions, corner cases, etc.

:::

-- NOTES --

-- N.1 -- Implicit string concatenation

:::
Inside all described data blocks implicid string concatenation
must NOT happen, i.e. must be disabled in the parser.
:::

-- N.2 -- Line continuation “\” and multi-line strings

:::
Currently line continuation works in “hard-coded” manner,
i.e. these two code samples behave differently:
:::

```
if 1:
    s = """\
    abc"""
    print(s)

if 1:
    if 1:
        s = """\
        abc"""
        print(s)
```

:::
in the second case variable “s” has different value than in the first case.
This behaviour can be a barrier to implementing some of the mentioned
features inside data blocks. Therefore it is suggested to change the behavior
of “\” inside the blocks, i.e. so that it removes not only the new line
but also removes the corresponding indentation whitespace.

It is suggested to introduce a dedicated block type for multi-line strings
with similar formatting rules (see 4.2).
:::