pythonimmediate

user202729

Jan 17, 2023

CONTENTS:

This package is a Python-TeX binding. It requires the `pythonimmediate` TeX package to be installed.

The package works both ways – you can run Python code from TeX, or TeX code from Python.

There are different sections of the documentation:

- *pythonimmediate.simple* – interface that "just works" for typical users of the `pythonimmediate` TeX package, to use Python coding from TeX, who does not know TeX inner details such as category codes.

  Note that this should be read in conjunction with the `pythonimmediate` TeX package documentation.

- Some properties of the parent TeX interpreter can be accessed from *default_engine*.

- The rest: contain functions to control the precise category codes of the tokens.

  Read *expand_once()* and *NTokenList* for some examples.

- See the documentation of *pythonimmediate.engine.ChildProcessEngine* for ways to create a TeX engine from inside Python, and explanation of the `engine=` optional argument for most functions.

- The command-line arguments that the Python component accepts (can be specified through the `args=` TeX module option) are documented in *pythonimmediate.pytotex*.

CHAPTER

ONE

PYTHONIMMEDIATE PACKAGE

1.1 Submodules

1.2 pythonimmediate.communicate module

class pythonimmediate.communicate.**Communicator**

Bases: ABC

character = None

abstract static is_available() → bool

abstract send(*data: bytes*) → None

Send data. This function is called in the textopy part.

abstract static setup() → tuple[*Communicator*, ListenForwarder]

Constructs an communicator object, that can be used to send information to this process.

The Communicator should be sent to the other process as part of the GlobalConfiguration object.

The ListenForwarder should be called in this process.

class pythonimmediate.communicate.**GlobalConfiguration**(*debug: 'int' = 0*, *communicator: 'Communicator' = None*, *sanity_check_extra_line: 'bool' = False*, *debug_force_buffered: 'bool' = False*, *naive_flush: 'bool' = False*)

Bases: object

communicator: *Communicator* = None

debug: int = 0

debug_force_buffered: bool = False

static from_args(*args: Namespace*, *communicator:* Communicator) → *GlobalConfiguration*

naive_flush: bool = False

sanity_check_extra_line: bool = False

class pythonimmediate.communicate.**MultiprocessingNetworkCommunicator**(*port: 'int'*, *connection: 'Optional[Connection]' = None*)

Bases: *Communicator*

connection: Optional[Connection] = None

static is_available() → bool

port: int

send(*data: bytes*) → None

Send data. This function is called in the textopy part.

static setup() → tuple[*MultiprocessingNetworkCommunicator*, ListenForwarder]

Constructs an communicator object, that can be used to send information to this process.

The Communicator should be sent to the other process as part of the GlobalConfiguration object.

The ListenForwarder should be called in this process.

class pythonimmediate.communicate.**UnnamedPipeCommunicator**(*pid: 'int'*, *fileno: 'int'*, *connection: 'Optional[IO[bytes]]' = None*)

Bases: *Communicator*

connection: Optional[IO[bytes]] = None

fileno: int

static is_available() → bool

pid: int

send(*data: bytes*) → None

Send data. This function is called in the textopy part.

static setup() → tuple[*UnnamedPipeCommunicator*, ListenForwarder]

Constructs an communicator object, that can be used to send information to this process.

The Communicator should be sent to the other process as part of the GlobalConfiguration object.

The ListenForwarder should be called in this process.

1.3 pythonimmediate.copy_to_stderr module

module to copy everything from stdin to stderr.

needed to allow TeX to write to stderr.

1.4 pythonimmediate.decode_8bit module

1.5 pythonimmediate.engine module

Abstract engine class.

class pythonimmediate.engine.**ChildProcessEngine**(*engine_name: Literal['pdftex', 'xetex', 'luatex']*, *args: Iterable[str] = ()*)

Bases: *Engine*

An object that represents a TEX engine that runs as a subprocess of this process.

Can be used as a context manager to automatically close the subprocess when the context is exited.

For example, the following Python code, if run alone, will spawn a TEX process and use it to write "Hello world" to a file named `a.txt` in the temporary directory:

```
from pythonimmediate.engine import ChildProcessEngine
from pythonimmediate import execute

with ChildProcessEngine("pdftex") as engine:
    # do something with the engine, for example:
    execute(r'''
    \immediate\openout9=a.txt
    \immediate\write9{Hello world}
    \immediate\closeout9
    ''', engine=engine)

# now the engine is closed.
```

Note that explicit `engine` argument must be passed in most functions.

See *DefaultEngine* for a way to bypass that.

close() → None

sent a `r` to the process so TeX exits gracefully.

this might be called from `__del__()` so do not import anything here.

get_process() → Popen

class pythonimmediate.engine.**DefaultEngine**

Bases: *Engine*

A convenience class that can be used to avoid passing explicit `engine` argument to functions.

This is not thread-safe.

Users should not instantiate this class directly. Instead, use *default_engine*.

Usage example:

```
default_engine.set_engine(engine)
execute("hello world")  # this is executed on engine=engine
```

See also:

set_engine()

property config: *GlobalConfiguration*

Self-explanatory.

engine: Optional[*Engine*]

Stores the engine being set internally.

Normally there's no reason to access the internal engine directly, as `self` can be used like the engine inside.

get_engine() → *Engine*

Convenience helper function, return the engine.

All the other functions that use this one (those that make use of the engine) will raise RuntimeError if the engine is None.

property name: Literal['pdftex', 'xetex', 'luatex']

Self-explanatory.

set_engine(*engine: Optional[*Engine*]*) → *SetDefaultEngineContextManager*

Set the default engine to another engine.

Can also be used as a context manager to revert to the original engine. Example:

```
with default_engine.set_engine(...):
    pass  # do something
# now the original engine is restored
```

class pythonimmediate.engine.**Engine**

Bases: ABC

check_not_exited(*message: str*) → None

Internal function.

check_not_exited_after() → None

Internal function.

check_not_exited_before() → None

Internal function.

check_not_finished() → None

Internal function.

property config: *GlobalConfiguration*

Self-explanatory.

property is_unicode: bool

Self-explanatory.

property name: Literal['pdftex', 'xetex', 'luatex']

Self-explanatory.

read() → bytes

Internal function.

Read one line from the engine.

It must not be EOF otherwise there's an error.

The returned line does not contain the newline character.

write(*s: bytes*) → None

class pythonimmediate.engine.**ParentProcessEngine**(*pseudo_config:* GlobalConfiguration)

Bases: *Engine*

Represent the engine if this process is started by the TeX's pythonimmediate library.

This should not be used directly. Only pythonimmediate.main module should use this.

class pythonimmediate.engine.**SetDefaultEngineContextManager**(*old_engine: Optional[*Engine*]*)

Bases: object

Context manager, used in conjunction with default_engine.set_engine(...) to revert to the original engine.

old_engine: Optional[*Engine*]

pythonimmediate.engine.**debug_possibly_shorten**(*line: str*) → str

pythonimmediate.engine.**default_engine = <pythonimmediate.engine.DefaultEngine object>**

A constant that can be used to avoid passing explicit engine argument to functions.

See documentation of *DefaultEngine* for more details.

For Python running inside a TeX process, useful attributes are *name* and *is_unicode*.

1.6 pythonimmediate.pytotex module

pythonimmediate.pytotex.**get_parser**() → ArgumentParser

1.7 pythonimmediate.simple module

Simple interface, suitable for users who may not be aware of TeX subtleties, such as category codes.

This is only guaranteed to work properly in normal category code situations (in other words, \makeatletter or \ExplSyntaxOn are not officially supported). In all other cases, use the advanced API.

Nevertheless, there are still some parts that you must be aware of:

- Any "output", resulted from any command related to Python, can only be used to typeset text, it must not be used to pass "values" to other TeX commands.

  This is already mentioned in the documentation of the TeX file in \py command, just repeated here.

  For example:

```
\py{1+1}  % legal, typeset 2

\setcounter{abc}{\py{1+1}}  % illegal

% the following is still illegal no many how many ``\expanded`` and such are used
\edef\tmp{\py{1+1}}
\setcounter{abc}{\tmp}

\py{ r'\\setcounter{abc}{' + str(1+1) + '}' }  % legal workaround
```

  Similar for commands defined with *newcommand()*:

```
@newcommand
def test():
    print_TeX("123", end="")
```

Then the TEX code:

```
The result is \test.  % legal, typesets "123"
\setcounter{abc}{\test}  % illegal
```

- There's the function *fully_expand()*, but some TEX commands cannot be used inside this.

  Refer to the *Note on unexpandable and fragile macros* section for more details.

- Regarding values being passed from TEX to Python, you can do one of the following:
  - Either simply do `\py{value = r'''#1'''}`, or
  - `\py{value = get_arg_str()}{#1}`.

  You can easily see both forms are equivalent, except that in the first form, `#1` cannot end with unbalanced `\` or contain triple single quotes.

Start with reading *newcommand()* and *execute()*. Then read the rest of the functions in this module.

pythonimmediate.simple.**define_char**(*char: str*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → Callable[[T2], T2]

Define a character to do some specific action.

Can be used as a decorator:

```
@define_char("×")
def multiplication_sign():
    print_TeX(end=r"\times")
```

Note: It's **not recommended** to define commonly-used characters, for example if you define `n` then commands whose name contains `n` cannot be used anymore.

As another example, if you define `-`, then commands like `\draw (1, 2) -- (3, 4);` in TikZ cannot be used.

undefine_char() can be used to undo the effect.

---

pythonimmediate.simple.**execute**(*block: str*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Run a block of TEX-code (might consist of multiple lines).

A simple example is `execute('123')` which simply typesets `123` (with a trailing newline, see the note in *newcommand()* documentation).

This is similar to *print_TeX()* but it's executed immediately, and any error is immediately reported with Python traceback points to the caller.

On the other hand, because this is executed immediately, each part must be "self-contained". With *print_TeX()* you can do the following:

```
print_TeX(r"\abc", end="")
print_TeX(r"def", end="")
```

and a single command `\abcdef` will be executed, but it does not work with this method.

As another consequence, all *execute()* are done before all *print_TeX()* in the same block. For example:

```
print_TeX(r"3")
execute(r"1")
print_TeX(r"4")
execute(r"2")
```

will typeset `1 2 3 4`.

Note: For advanced users: it's implemented with `\scantokens`, so catcode-changing commands are allowed inside.

---

pythonimmediate.simple.**f1**(*s: str*, *globals: Optional[dict] = None*, *locals: Optional[dict] = None*, *escape: Optional[Union[Tuple[str, ...], str]] = None*) → str

Helper function to construct a string from Python expression parts, similar to f-strings, but allow using arbitrary delimiters.

This is useful for constructing TEX code, because the { and } characters are frequently used in TEX.

Example:

```
>>> a = 1
>>> b = 2
>>> f1("a=`a`, b=`b`")
'a=1, b=2'
>>> f1("``")
Traceback (most recent call last):
    ...
ValueError: Unbalanced delimiters!
```

The escape/delimiter character can be escaped by doubling it:

```
>>> f1("a``b")
'a`b'
>>> f1("a ` '````' ` b")
'a `` b'
```

It's possible to customize the delimiters:

```
>>> f1("a=!a!", escape="!")
'a=1'
```

It's possible to use different delimiters for the opening and the closing:

```
>>> f1("a=a, b=b", escape="")
'a=1, b=2'
>>> f1("<", escape="<>")
Traceback (most recent call last):
    ...
ValueError: Unbalanced delimiters!
```

There's no way to escape them, they must be balanced in the string:

```
>>> f1("a= ''+str(a)+'' ", escape="")
'a=1'
```

The default value of *escape* is stored in `f1.default_escape`, it can be reassigned:

```
>>> f1.default_escape="!"
>>> f1("a=!a!")
'a=1'
>>> f1.default_escape="`"
```

It's even possible to use multiple characters as the delimiter:

```
>>> f1("a=**a**, b=**b**", escape=["**"])
'a=1, b=2'
>>> f1("a=[[a]], b=[[b]]", escape=["[[", "]]"])
'a=1, b=2'
```

In the first case above, remember to put the delimiter in a list because otherwise it will be interpreted as an opening and a closing delimiter.

Currently, format specifiers such as `:d` are not supported.

pythonimmediate.simple.**fully_expand**(*content: str*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Expand all macros in the given string.

1.7.1 Note on unexpandable and fragile macros

Not all TEX commands/macros can be used inside this function. If you try to, they will either return the input unchanged, or error out.

In such cases, the problem is **impossible** to solve, so just look for workarounds.

An example is the `\py` command, which is already mentioned in the initial introduction to the *simple* module:

```
>> fully_expand(r"\py{1+1}")
r"\py{1+1}"
```

Some other examples (assuming `\usepackage{ifthen}` is used):

```
>> execute(r'\ifthenelse{1=2}{equal}{not equal}')  # legal, typeset "not equal"
>> fully_expand(r'\ifthenelse{1=2}{equal}{not equal}')  # error out

>> execute(r'\uppercase{abc}')  # legal, typeset "ABC"
>> fully_expand(r'\uppercase{abc}')  # just returned verbatim
r'\uppercase {abc}'
```

There might be a few workarounds, but they're all dependent on the exact macros involved.

For if-style commands, assign the desired result to a temporary variable:

```
>> execute(r'\ifthenelse{1=2}{\pyc{result = "equal"}}{\pyc{result = "not equal"}}')
>> result
"not equal"
```

For for-loop-style commands, you can do something similar (demonstrated here with `tikz`'s `\foreach` command):

```
>> result = []
>> execute(r'\foreach \x in {1, 2, ..., 10} {\pyc{result.append(var["x"])}}')
>> result
['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']
```

For macros such as `\uppercase{...}`, there's no simple workaround. Reimplement it yourself (such as with `.upper()` in Python).

pythonimmediate.simple.**get_arg_estr**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Get a mandatory argument from the input stream, then process it as described in *Note on argument expansion of estr-type functions*.

1.7.2 Note on argument expansion of `estr`-type functions

These functions return a string, however they're meant for strings other than TeX code.

The argument is fully expanded, and any "escaped character" are passed as the character itself to Python.

The behavior is similar to that of the `\py` command argument processing, see also the TeX package documentation.

Some examples: After the TeX code `\def\myvalue{123abc}` is executed, then:

- if the argument in TeX code is `{abc}`, then the Python function will return `"abc"` (similar to as if *get_arg_str()* is used),
- if the argument in TeX code is `{\%a\#b\\\~}`, then the Python function will return `r"%a#b\~"`,
- if the argument in TeX code is `{\myvalue}`, then the Python function will return `"123abc"`.

pythonimmediate.simple.**get_arg_str**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Get a mandatory argument from the input stream.

It's slightly difficult to explain what this does, so here's an example:

```
@newcommand
def foo():
    a = get_arg_str()
    b = get_arg_str()
    c = get_arg_str()
```

The above defines a TeX function `\foo`, if it's called in TeX code as:

```
\foo{xx}{yy}{zz}
```

then the variable `a` will be the string `"xx"`, `b` will be `"yy"`, and `c` will be `"zz"`.

1.7.3 Note on parameter tokenization

Note: This function, as well as all the *_str* functions, return the argument that might be mangled in the following ways:

- A space might be added after each command that consist of multiple characters
- the `^^xx` or `^^xxxx` etc. notation is replaced by the character itself, or vice versa – literal tab character might get replaced with `^^I`
- multiple space characters may be collapsed into one
- newline character may become space character
- double-newline character may become `\par`

For example, `\*\hell^^6F{ }` may become the string `r"\*\hello { }"` in Python.

As such, verbatim-like commands in the argument are not supported. See *get_verb_arg()* for a workaround.

Nevertheless, if the argument consist of only "normal" TEX commands, re-executing the string should do the same thing as the original code.

In case of doubt, print out the string and check it manually.

Note: For advanced users:

This function corresponds to the `m`-type argument in `xparse` package.

It gets the argument, detokenize it, pass it through `_replace_double_hash()`, and return the result.

This is the simple API, as such it assumes normal category code values. Refer to `BalancedTokenList.get_next()` for a more advanced API.

pythonimmediate.simple.**get_multiline_verb_arg**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Get a multi-line verbatim argument. Usage is identical to *get_verb_arg()*, except newline characters in the argument is supported.

Note: in unusual category regime (such as that in `\ExplSyntaxOn`), it may return wrong result.

pythonimmediate.simple.**get_next_char**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Return the character of the following token as with *peek_next_char()*, but also removes it from the input stream.

pythonimmediate.simple.**get_optional_arg_estr**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → Optional[str]

Get an optional argument. See also *Note on argument expansion of estr-type functions*.

pythonimmediate.simple.**get_optional_arg_str**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → Optional[str]

Get an optional argument. See also *Note on parameter tokenization*.

Note: For advanced users: This function corresponds to the o-type argument in `xparse` package.

pythonimmediate.simple.**get_verb_arg**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Get a verbatim argument.

Similar to `\verb`, defined TEX commands that use this function can only be used at top level i.e. not inside any arguments.

This function behavior is identical to v-type argument in `xparse` package, you can use it like this:

```
\foo{xx%^\?}  % delimited with braces
\foo|xx%^\?|  % delimited with another symbol
```

See also:

get_multiline_verb_arg() to support newline character in the argument.

Note: Hard TAB character in the argument gives an error until the corresponding LaTeX3 bug is fixed, see https://tex.stackexchange.com/q/508001/250119.

pythonimmediate.simple.**is_balanced**(*content: str*) → bool

Check if the given string is balanced, i.e. if all opening and closing braces match.

This is a bit tricky to implement, it's recommended to use the library function.

For example:

```
>>> is_balanced("a{b}c")
True
>>> is_balanced("a{b}c}")
False
>>> is_balanced(r"\{")
True
```

pythonimmediate.simple.**newcommand**(*name: str*, *f: Optional[Callable]*, *engine:* Engine) → Callable

Define a new TEX-command.

Example:

```
@newcommand
def myfunction():
    print_TeX("Hello world!", end="")
```

The above is mostly equivalent to `\newcommand{\myfunction}{Hello world!}`, it defines a TEX command `\myfunction` that prints `Hello world!`.

See also documentation of *print_TeX()* to understand the example above.

It can be used as either a decorator or a function:

```
def myfunction():
    print_TeX("Hello world!", end="")

newcommand("myfunction", myfunction)
```

An explicit command name can also be provided:

```
@newcommand("hello")
def myfunction():
    print_TeX("Hello world!", end="")
```

The above defines a TeX command \hello that prints Hello world!.

If name is not provided, it's automatically deduced from the Python function name.

The above is not by itself very useful. Read the documentation of the following functions sequentially for a guide on how to define commands that take arguments:

- *get_arg_str()*
- *get_arg_estr()*
- *get_verb_arg()*
- *get_optional_arg_str()*
- *peek_next_char()*
- *get_next_char()*

Then see also (as natural extensions of the above):

- *get_multiline_verb_arg()*
- *get_optional_arg_estr()*

1.7.4 Note on trailing newline

Note: Regarding the end="" part in the example above, it's used to prevent a spurious space, otherwise if you run the TeX code:

```
123\myfunction 456
```

it will be "equivalent" to:

```
123Hello world!
456
```

and the newline inserts a space.

Internally, the feature is implemented by appending % to the last line, as most of the time the code:

```
123Hello world!%
456
```

will be equivalent to:

```
123Hello world!456
```

but in unusual category code cases or cases of malformed TeX code (such as a trailing backslash at the end of the line), it may not be equivalent. Use the advanced API, or always include a final newline and manually add the comment character, if these cases happen.

---

pythonimmediate.simple.**newenvironment**(*name: str*, *f: Optional[Callable]*, *engine:* Engine) → Callable

Define a new TeX normal environment.

Note that the environment will normally not have access to the body of the environment, see *newenvironment_verb()* for some alternatives.

Parameters

- **name** – the name of the environment, e.g. "myenv" or "myenv*".
- **f** – a function that should execute the code for the begin part of the environment, yield, then execute the code for the end part of the environment.
- **engine** – the engine to use.

Usage example:

```
@newenvironment("myenv")
def myenv():
    x=random.randint(1, 100)
    print_TeX(f"begin {x}")
    yield
    print_TeX(f"end {x}")
```

then the following TeX code:

```
\begin{myenv}
\begin{myenv}
Hello world!
\end{myenv}
\end{myenv}
```

might typeset the following content:

```
begin 42
begin 24
Hello world!
end 24
end 42
```

Functions such as *get_arg_str()* etc. can also be used in the first part of the function to take arguments.

If the name is omitted, the function name is used as the environment name.

It can be used either as a decorator or a function, see *newcommand()* for details.

pythonimmediate.simple.**newenvironment_verb**(*name: str*, *f: Optional[Callable]*, *engine:* Engine) → Callable

Define a new TeX environment that reads its body verbatim.

Note that space characters at the end of each line are removed.

The environment must not take any argument. For example the following built-in tabular environment takes some argument, so cannot be implemented with this function:

```
\begin{tabular}[t]
    ...
\end{tabular}
```

It can be used either as a decorator or a function, see *newenvironment()* for details.

Some usage example:

```
@newenvironment_verb("myenv")
def myenv(body: str):
    execute(body.replace("a", "b"))
```

If later the following TEX code is executed:

```
\begin{myenv}
aaa
\end{myenv}
```

then the value of the variable body will be "aaa\n", and the following content will be typeset:

```
bbb
```

Note: For advanced users: unlike a typical environment, the code will not be executed in a new *TEX group*.

pythonimmediate.simple.**peek_next_char**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Get the character of the following token, or empty string if it's not a character.

This function can be used as follows to simulate e-type argument specifiers of xparse:

```
if peek_next_char()=="^":
    get_next_char()
    result = get_arg_str()
    # the following content in the input stream is ``^{...}`` and we stored the␣
↪content inside to ``result``
else:
    pass  # there's no ``^...`` following in the input stream
```

It can also simulate s-type argument specifier (optional star):

```
if peek_next_char()=="*":
    get_next_char()
    # there's a star
else:
    pass  # there's no star
```

It can even simulate o-type argument specifier (optional argument delimited by [...]):

```
if peek_next_char()=="[":
    get_next_char()  # skip the `[`
    result=""
    while True:
        if peek_next_char():
            c=get_next_char()
            if c=="]": break
            result+=c
        else:
            # following in the input stream must be a control sequence, such as `\
```

(continues on next page)

(continued from previous page)

```
↪relax`
            result+=get_arg_str()
    # now result contains the content inside the `[...]`
else:
    pass  # there's no optional argument
```

Note that the above does not take into account the balance of braces or brackets, so:

- If the following content in the input is `[ab{cd]ef}gh]` then the result will be `ab{cd`.
- If the following content in the input is `[ab[cd]ef]` then the result will be `ab[cd`.

Note: For advanced users:

This function uses `peek_next_meaning()` under the hood to get the meaning of the following token. See that function documentation for a warning on undefined behavior.

Will also return nonempty if the next token is an implicit character token. This case is not supported and might give random error.

`pythonimmediate.simple.`**`print_TeX`**`(`**args*, ***kwargs*`)` → None

Unlike other packages, the normal `print()` function prints to the console.

This function can be used to print TEX code to be executed. For example:

```
\begin{pycode}
print_TeX("Hello world!")
\end{pycode}
```

It can also be used inside a custom-defined command, see the documentation of *newcommand()* for an example.

The signature is identical to that of `print()`. Passing explicit `file=` argument is not allowed.

Remark: normally the `print()` function prints out a newline with each invocation. A newline will appear as a spurious space in the output Use `print_TeX(end="")` to prevent this, see *Note on trailing newline*.

`pythonimmediate.simple.`**`put_next`**`(`*arg: str*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*`)` → None

Put some content forward in the input stream.

Parameters

`arg` – The content, must be a single line.

Note that there must not be any verbatim-like commands in the argument, so for example `put_next(r"\verb|a|")` is not allowed.

If there might be verbatim-like arguments, the problem is (almost) unsolvable. Refer to *print_TeX()* or *execute()* for workarounds, or use the advanced interface such as *pythonimmediate.BalancedTokenList.put_next()*.

For example, if the following content in the input stream are `{abc}{def}`:

```
s = get_arg_str()  # s = "abc"
t = get_arg_str()  # t = "def"
put_next("{" + t + "}")
put_next("{" + s + "}")
```

After the above code, the content in the input stream is "mostly unchanged".

Note: For advanced users: the argument is tokenized in the current category regime.

pythonimmediate.simple.**renewcommand**(*name: str*, *f: Optional[Callable]*, *engine:* Engine) → Callable

Redefine a TEX-command. Usage is similar to *newcommand()*.

pythonimmediate.simple.**run_tokenized_line_local**(*line: str*, *, *check_braces: bool = True*, *check_newline: bool = True*, *check_continue: bool = True*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

pythonimmediate.simple.**split_balanced**(*content: str*, *separator: str*) → List[str]

Split the given string at the given substring, but only if the parts are balanced.

This is a bit tricky to implement, it's recommended to use the library function.

If either *content* or *separator* is unbalanced, the function will raise ValueError.

For example:

```
>>> split_balanced("a{b,c},c{d}", ",")
['a{b,c}', 'c{d}']
>>> split_balanced(r"\{,\}", ",")
['\\{', '\\}']
>>> split_balanced("{", "{")
Traceback (most recent call last):
    ...
ValueError: Content is not balanced!
```

pythonimmediate.simple.**strip_optional_braces**(*content: str*) → str

Strip the optional braces from the given string, if the whole string is wrapped in braces.

For example:

```
>>> strip_optional_braces("{a}")
'a'
>>> strip_optional_braces("a")
'a'
>>> strip_optional_braces("{a},{b}")
'{a},{b}'
```

pythonimmediate.simple.**undefine_char**(*char: str*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

The opposite of *define_char()*.

pythonimmediate.simple.**var = VarManager(engine=<pythonimmediate.engine.DefaultEngine object>)**

Get and set value of "variables" in TEX.

Can be used like this:

```
var["myvar"]="myvalue"  # "equivalent" to \def\myvar{myvalue}
var["myvar"]=r"\textbf{123}"  # "equivalent" to \def\myvar{\textbf{123}}
```

(continues on next page)

(continued from previous page)

```
var["myvar"]=r"\def\test#1{#1}"  # "equivalent" to \tl_set:Nn \myvar {\def\test#1{
↪#1}}  (note that `#` doesn't need to be doubled here unlike in `\def`)
var(engine)["myvar"]="myvalue"  # pass explicit engine

# after ``\def\myvar{myvalue}`` (or ``\edef``, etc.) on TeX you can do:
print(var["myvar"])  # get the value of the variable, return a string
```

It's currently undefined behavior if the passed-in value is not a "variable". (in some future version additional checking might be performed in debug run)

Notes in *Note on parameter tokenization* apply – in other words, after you set a value it may become slightly different when read back.

1.8 pythonimmediate.textopy module

pythonimmediate.textopy.**main**() → None

1.9 Module contents

class pythonimmediate.**BalancedTokenList**(*a: Iterable = ()*)

Bases: *TokenList*

Represents a balanced token list.

Some useful methods to interact with TEX include *expand_o()*, *expand_x()*, *get_next()* and *put_next()*. See the corresponding methods' documentation for usage examples.

See also *Token list construction* for shorthands to construct token lists in Python code.

Note: Runtime checking is not strictly enforced, use *is_balanced()* method explicitly if you need to check.

---

detokenize(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Returns
a string, equal to the result of `\detokenize` applied to this token list.

execute(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Execute this token list. It must not "peek ahead" in the input stream.

For example the token list `\catcode1=2\relax` can be executed safely (and sets the corresponding category code), but there's no guarantee what will be assigned to `\tmp` when `\futurelet\tmp` is executed.

expand_o(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → *BalancedTokenList*

Return the o-expansion of this token list.

The result must be balanced, otherwise the behavior is undefined.

expand_x(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → *BalancedTokenList*

Return the x-expansion of this token list.

The result must be balanced, otherwise the behavior is undefined.

static get_next(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → *BalancedTokenList*

Get an (undelimited) argument from the TEX input stream.

put_next(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Put this token list forward in the input stream.

class pythonimmediate.**BlueToken**(*token:* Token)

Bases: *NToken*

Represents a blue token (see documentation of *NToken*).

property no_blue: *Token*

Return the result of this token after being "touched", which drops its blue status if any.

property noexpand: *BlueToken*

Return the result of \noexpand applied on this token.

put_next(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Put this token forward in the input stream.

repr1() → str

token: *Token*

pythonimmediate.**C**

alias of *Catcode*

class pythonimmediate.**Catcode**(*value*)

Bases: Enum

This class contains a shorthand to allow creating a token with little Python code. The individual *Catcode* objects can be called with either a character or a character code to create the object:

```
Catcode.letter("a")  # creates a token with category code letter and character code
↪"a"=chr(97)
Catcode.letter(97)  # same as above
```

Both of the above forms are equivalent to `CharacterToken(index=97, catcode=Catcode.letter)`.

See also *Token list construction* for more ways of constructing token lists.

active = 13

alignment = 4

begin_group = 1

bgroup = 1

comment = 14

egroup = 2

end_group = 2

end_of_line = 5

escape = 0

property for_token: bool

Return whether a *CharacterToken* may have this catcode.

ignored = 9

invalid = 15

letter = 11

line = 5

math = 3

math_subscript = 8

math_superscript = 7

math_toggle = 3

other = 12

paragraph = 5

param = 6

parameter = 6

space = 10

subscript = 8

superscript = 7

class pythonimmediate.**CharacterToken**(*index: 'int'*, *catcode: 'Catcode'*)

Bases: *Token*

property assignable: bool

Whether this token can be assigned to i.e. it's control sequence or active character.

property can_blue: bool

Return whether this token can possibly be blue i.e. expandable.

catcode: *Catcode*

property chr: str

The character of this token. For example `Catcode.letter("a").chr=="a"`.

degree() → int

return the imbalance degree for this token ({ -> 1, } -> -1, everything else -> 0)

index: int

The character code of this token. For example `Catcode.letter("a").index==97`.

repr1() → str

serialize() → str

Internal function, serialize this token to be able to pass to TEX.

simple_detokenize(*get_catcode: Callable[[int],* Catcode*]*) → str

Simple approximate detokenizer, implemented in Python.

str_unicode() → str

`self` must represent a character of a TEX string. (i.e. equal to itself when detokenized)

Returns
the string content.

Note: See *NTokenList.str_unicode()*.

---

class pythonimmediate.**ControlSequenceToken**(*csname: str*)

Bases: *Token*

Represents a control sequence.

Note that currently, on non-Unicode engines, the `csname` field is represented in a particular way: each character represents a byte in the TokenList, and thus it has character code no more than 255.

So for example, the control sequence obtained by expanding `\csname \endcsname` once has `.csname` field equal to `"\xe2\x84\x9d"` (which has `len=3`).

property assignable: bool

Whether this token can be assigned to i.e. it's control sequence or active character.

can_blue = True

csname: str

make = <pythonimmediate.ControlSequenceTokenMaker object>

Refer to the documentation of *ControlSequenceTokenMaker*.

repr1() → str

serialize() → str

Internal function, serialize this token to be able to pass to TEX.

simple_detokenize(*get_catcode: Callable[[int],* Catcode*]*) → str

Simple approximate detokenizer, implemented in Python.

class pythonimmediate.**ControlSequenceTokenMaker**(*prefix: str*)

Bases: `object`

Shorthand to create control sequence objects in Python easier.

There's a default one that can be used as, if you assign `T=ControlSequenceToken.make`, then `T.hello` returns the token `\hello`.

class pythonimmediate.**FrozenRelaxToken**

Bases: *Token*

assignable = False

can_blue = False

repr1() → str

serialize() → str

Internal function, serialize this token to be able to pass to TEX.

simple_detokenize(*get_catcode: Callable[[int],* Catcode*]*) → str

Simple approximate detokenizer, implemented in Python.

class pythonimmediate.**NToken**

Bases: ABC

Represent a possibly-notexpanded token. For convenience, a notexpanded token is called a blue token. It's not always possible to determine the notexpanded status of a following token in the input stream.

Implementation note: Token objects must be frozen.

degree() → int

return the imbalance degree for this token ({ -> 1, } -> -1, everything else -> 0)

meaning_equal(*other: NToken*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → bool

Whether this token is the same in meaning as the token specified in the parameter *other*.

Note that two tokens might have different meaning despite having equal *meaning_str()*.

meaning_str(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

get the meaning of this token as a string.

Note that all blue tokens have the meaning equal to \relax (or [unknown command code! (0, 1)] in a buggy LuaTeX implementation) with the backslash replaced by the current escapechar.

abstract property no_blue: *Token*

Return the result of this token after being "touched", which drops its blue status if any.

abstract property noexpand: *NToken*

Return the result of \noexpand applied on this token.

abstract put_next(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*)

Put this token forward in the input stream.

abstract repr1() → str

str_unicode() → str

self must represent a character of a TEX string. (i.e. equal to itself when detokenized)

Returns
the string content.

Note: See *NTokenList.str_unicode()*.

class pythonimmediate.**NTokenList**(*a: Iterable = ()*)

Bases: UserList

Similar to *TokenList*, but can contain blue tokens.

The class can be used identical to a Python list consist of *NToken* objects, plus some additional methods to operate on token lists.

Refer to the documentation of *TokenList* for some usage example.

bool() → bool

Internal function. `self` must represent a TEX string either equal to "0" or "1".

Returns
the boolean represented by the string.

execute(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

See *BalancedTokenList.execute()*.

expand_x(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → *BalancedTokenList*

See *BalancedTokenList.expand_x()*.

static force_token_list(*a: Iterable*) → Iterable[*NToken*]

is_balanced() → bool

Check if this is balanced.

put_next(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

See *BalancedTokenList.put_next()*.

simple_parts() → List[Union[*BalancedTokenList*, *Token*, *BlueToken*]]

Internal function.

Split this *NTokenList* into a list of balanced non-blue parts, unbalanced {/} tokens, and blue tokens.

str(*engine:* Engine) → str

`self` must represent a TEX string. (i.e. equal to itself when detokenized)

Returns
the string content.

str_unicode() → str

`self` must represent a TEX string. (i.e. equal to itself when detokenized)

Returns
the string content.

Note: In non-Unicode engines, each token will be replaced with a character with character code equal to the character code of that token. UTF-8 characters with character code >=`0x80` will be represented by multiple characters in the returned string.

---

class pythonimmediate.**PTTBalancedTokenList**(*data: 'BalancedTokenList'*)

Bases: *PyToTeXData*

data: *BalancedTokenList*

read_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

serialize(*engine:* Engine) → bytes

Return a bytes object that can be passed to `engine.write()` directly.

class pythonimmediate.**PTTBlock**(*data: 'str'*)

Bases: *PyToTeXData*

data: **str**

read_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

serialize(*engine:* Engine) → bytes

Return a bytes object that can be passed to `engine.write()` directly.

class pythonimmediate.**PTTInt**(*data: 'int'*)

Bases: *PyToTeXData*

data: **int**

read_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

serialize(*engine:* Engine) → bytes

Return a bytes object that can be passed to `engine.write()` directly.

class pythonimmediate.**PTTTeXLine**(*data: str*)

Bases: *PyToTeXData*

Represents a line to be tokenized in TeX's current catcode regime. The trailing newline is not included, i.e. it's tokenized under `\endlinechar=-1`.

data: **str**

read_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

serialize(*engine:* Engine) → bytes

Return a bytes object that can be passed to `engine.write()` directly.

class pythonimmediate.**PTTVerbatimLine**(*data: 'str'*)

Bases: *PyToTeXData*

data: **str**

read_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

serialize(*engine:* Engine) → bytes

Return a bytes object that can be passed to `engine.write()` directly.

class pythonimmediate.**PTTVerbatimRawLine**(*data: bytes*)

Bases: *PyToTeXData*

Represents a line to be tokenized verbatim. Internally the `\readline` primitive is used, as such, any trailing spaces are stripped. The trailing newline is not included, i.e. it's read under `\endlinechar=-1`.

data: bytes

read_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

serialize(*engine:* Engine) → bytes

Return a bytes object that can be passed to `engine.write()` directly.

class pythonimmediate.**PyToTeXData**

Bases: ABC

Internal class (for now). Represent a data type that can be sent from Python to $\TeX$.

abstract static read_code(*var: str*) → str

Takes an argument, the variable name (with backslash prefixed such as `"\abc"`.)

Returns
some $\TeX$ code that when executed in expl3 category code regime,

will read a value of the specified data type and assign it to the variable.

abstract serialize(*engine:* Engine) → bytes

Return a bytes object that can be passed to `engine.write()` directly.

class pythonimmediate.**PythonCallTeXFunctionType**(**args*, ***kwargs*)

Bases: Protocol

class pythonimmediate.**PythonCallTeXSyncFunctionType**(**args*, ***kwargs*)

Bases: *PythonCallTeXFunctionType*, Protocol

class pythonimmediate.**Python_call_TeX_data**(*TeX_code: 'str'*, *recursive: 'bool'*, *finish: 'bool'*, *sync: 'Optional[bool]'*)

Bases: object

TeX_code: str

finish: bool

recursive: bool

sync: Optional[bool]

class pythonimmediate.**Python_call_TeX_extra**(*ptt_argtypes: 'Tuple[Type[PyToTeXData], ...]'*, *ttp_argtypes: 'Union[Type[TeXToPyData], Tuple[Type[TeXToPyData], ...]]'*)

Bases: object

ptt_argtypes: Tuple[Type[*PyToTeXData*], ...]

ttp_argtypes: Union[Type[*TeXToPyData*], Tuple[Type[*TeXToPyData*], ...]]

pythonimmediate.**Python_call_TeX_local**(*TeX_code: str*, *, *recursive: bool = True*, *sync: Optional[bool] = None*, *finish: bool = False*) → Callable

Internal function. See *scan_Python_call_TeX()*.

class pythonimmediate.**RedirectPrintTeX**(*t*)

Bases: object

class pythonimmediate.**TTPBalancedTokenList**(*a: Iterable = ()*)

Bases: *TeXToPyData*, *BalancedTokenList*

static read(*engine:* Engine) → *TTPBalancedTokenList*

Given that TEX has just sent the data, read into a Python object.

send_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces (‘{‘ and ‘}’).

send_code_var()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces (‘{‘ and ‘}’).

class pythonimmediate.**TTPBlock**

Bases: *TeXToPyData*, str

static read(*engine:* Engine) → *TTPBlock*

Given that TEX has just sent the data, read into a Python object.

send_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces (‘{‘ and ‘}’).

send_code_var()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces (‘{‘ and ‘}’).

class pythonimmediate.**TTPEBlock**

Bases: *TeXToPyData*, str

A kind of argument that interprets “escaped string” and fully expand anything inside. For example, {\} sends a single backslash to Python, {{} sends a single ‘{‘ to Python. Done by fully expand the argument in escapechar=-1 and convert it to a string. Additional precaution is needed, see the note above.

static read(*engine:* Engine) → *TTPEBlock*

Given that TEX has just sent the data, read into a Python object.

send_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces (‘{‘ and ‘}’).

send_code_var()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

class pythonimmediate.**TTPELine**

Bases: *TeXToPyData*, str

Same as TTPEBlock, but for a single line only.

static read(*engine:* Engine) → *TTPELine*

Given that TEX has just sent the data, read into a Python object.

send_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

send_code_var()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

class pythonimmediate.**TTPEmbeddedLine**

Bases: *TeXToPyData*, str

static read(*engine:* Engine) → *TTPEmbeddedLine*

Given that TEX has just sent the data, read into a Python object.

static send_code(*self*) → str

Return some TEX code that sends the argument to Python, where *arg* represents a token list or equivalent (such as #1).

static send_code_var(*self*) → str

Return some TEX code that sends the argument to Python, where *var* represents a token list variable (such as \l__my_var_tl) that contains the content to be sent.

class pythonimmediate.**TTPLine**

Bases: *TeXToPyData*, str

static read(*engine:* Engine) → *TTPLine*

Given that TEX has just sent the data, read into a Python object.

send_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

send_code_var()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

class pythonimmediate.**TTPRawLine**

Bases: *TeXToPyData*, bytes

static read(*engine:* Engine) → *TTPRawLine*

Given that TEX has just sent the data, read into a Python object.

send_code()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

send_code_var()

S.format(*args, **kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

class pythonimmediate.**TeXToPyData**

Bases: ABC

Internal class (for now). Represent a data type that can be sent from TEX to Python.

abstract static read(*engine:* Engine) → *TeXToPyData*

Given that TEX has just sent the data, read into a Python object.

abstract static send_code(*arg: str*) → str

Return some TEX code that sends the argument to Python, where *arg* represents a token list or equivalent (such as #1).

abstract static send_code_var(*var: str*) → str

Return some TEX code that sends the argument to Python, where *var* represents a token list variable (such as \l__my_var_tl) that contains the content to be sent.

class pythonimmediate.**Token**

Bases: *NToken*

Represent a TEX token, excluding the notexpanded possibility. See also documentation of *NToken*.

assign(*other: NToken*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Assign the meaning of this token to be equivalent to that of the other token.

assign_future(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Assign the meaning of this token to be equivalent to that of the following token in the input stream.

For example if this token is \a, and the input stream starts with bcde, then \a's meaning will be assigned to that of the explicit character b.

Note: Tokenizes one more token in the input stream, and remove its blue status if any.

assign_futurenext(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Assign the meaning of this token to be equivalent to that of the second-next token in the input stream.

For example if this token is \a, and the input stream starts with bcde, then \a's meaning will be assigned to that of the explicit character c.

Note: Tokenizes two more tokens in the input stream, and remove their blue status if any.

assign_value(*content:* BalancedTokenList) → None

Given `self` is an expl3 `tl`-variable, assign *content* to it locally.

abstract property assignable: bool

Whether this token can be assigned to i.e. it's control sequence or active character.

property blue: ***BlueToken***

Return a *BlueToken* containing `self`. *can_blue* must be true.

abstract property can_blue: bool

Return whether this token can possibly be blue i.e. expandable.

static deserialize(*s: str | bytes*) → *Token*

See documentation of *TokenList.deserialize()*.

Always return a single token.

static deserialize_bytes(*data: bytes*, *engine:* Engine) → *Token*

See documentation of *TokenList.deserialize_bytes()*.

Always return a single token.

static get_next(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → *Token*

Get the following token.

Note: in LaTeX3 versions without the commit https://github.com/latex3/latex3/commit/24f7188904d6 sometimes this may error out.

Note: because of the internal implementation of `\peek_analysis_map_inline:n`, this may tokenize up to 2 tokens ahead (including the returned token), as well as occasionally return the wrong token in unavoidable cases.

---

property no_blue: ***Token***

Return the result of this token after being "touched", which drops its blue status if any.

property noexpand: ***NToken***

Return the result of `\noexpand` applied on this token.

static peek_next(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → *Token*

Get the following token without removing it from the input stream.

Equivalent to *get_next()* then *put_next()* immediately. See documentation of *get_next()* for some notes.

put_next(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Put this token forward in the input stream.

abstract serialize() → str

Internal function, serialize this token to be able to pass to $\TeX$.

abstract simple_detokenize(*get_catcode: Callable[[int],* Catcode*]*) → str

Simple approximate detokenizer, implemented in Python.

value(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → *BalancedTokenList*

given `self` is a expl3 `tl`-variable, return the content.

value_str(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

given `self` is a expl3 `str`-variable, return the content.

class pythonimmediate.**TokenList**(*a: Iterable = ()*)

Bases: `UserList`

Represent a TEX token list, none of which can contain a blue token.

The class can be used identical to a Python list consist of *Token* objects, plus some additional methods to operate on token lists.

The list of tokens represented by this class does not need to be balanced. Usually you would want to use *BalancedTokenList* instead.

1.9.1 Token list construction

There are some functions to quickly construct token lists, see *from_string()* and `__init__()`.

`__init__()` is the constructor of the class, for a *TokenList* it can be used as follows:

- Given an existing token list, construct a copy (this usage is identical to that of the Python list constructor):

```
a=TokenList.doc("hello world")
b=TokenList(a)
```

- Construct a token list from a list of tokens:

```
a=TokenList([Catcode.letter("a"), Catcode.other("b"), T.test])
```

The above will define `a` to be `ab\test`, provided `T` is the object referred to in *ControlSequenceTokenMaker*.

See also *Catcode* for the explanation of the `Catcode.letter("a")` form.

- Construct a token list, using lists to represent nesting levels:

```
a=TokenList([T.edef, T.a, [T.x, T.y]])
```

The above creates a token list with content `\edef\a{\x\y}`.

The constructor of other classes such as *BalancedTokenList* and *NTokenList* works the same way.

property balanced: *BalancedTokenList*

`self` must be balanced.

Returns
a *BalancedTokenList* containing the content of this object.

balanced_parts() → List[Union[*BalancedTokenList*, *Token*]]

Internal function, used for serialization and sending to TEX.

Split this *TokenList* into a list of balanced parts and unbalanced `{/}` tokens.

bool() → bool

See *NTokenList.bool()*.

check_balanced() → None

ensure that this is balanced.

Raises
ValueError – if this is not balanced.

classmethod deserialize(*data: str | bytes*) → TokenListType

classmethod deserialize_bytes(*data: bytes*, *engine:* Engine) → TokenListType

Internal function.

Given a bytes object read directly from the engine, deserialize it.

classmethod doc(*s: str*) → TokenListType

Approximate tokenizer in document (normal) catcode regime.

Refer to documentation of *from_string()* for details.

Usage example:

```
BalancedTokenList.doc(r'\def\a{b}')  # returns an instance of BalancedTokenList␣
↪with the expected content
BalancedTokenList.doc('}')  # raises an error
TokenList.doc('}')  # returns an instance of TokenList with the expected content
```

classmethod e3(*s: str*) → TokenListType

Approximate tokenizer in expl3 (\ExplSyntaxOn) catcode regime.

Refer to documentation of *from_string()* for details.

Usage example:

```
BalancedTokenList.e3(r'\cs_new_protected:Npn \__mymodule_myfunction:n #1 { #1
↪#1 }')
# returns an instance of BalancedTokenList with the expected content
```

execute(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Execute this token list. It must not "peek ahead" in the input stream.

For example the token list \catcode1=2\relax can be executed safely (and sets the corresponding category code), but there's no guarantee what will be assigned to \tmp when \futurelet\tmp is executed.

expand_x(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → *BalancedTokenList*

Return the x-expansion of this token list.

The result must be balanced, otherwise the behavior is undefined.

static force_token_list(*a: Iterable*) → Iterable[*Token*]

classmethod from_string(*s: str*, *get_catcode: Callable[[int],* Catcode*]*) → TokenListType

Approximate tokenizer implemented in Python.

Convert a string to a *TokenList* (or some subclass of it such as *BalancedTokenList* approximately.

This is an internal function and should not be used directly. Use one of *e3()* or *doc()* instead.

These are used to allow constructing a *TokenList* object in Python without being too verbose. Refer to *Token list construction* for alternatives.

The tokenization algorithm is slightly different from TEX's in the following respect:

- multiple spaces are collapsed to one space, but only if it has character code space (32). i.e. in expl3 catcode, ~~ get tokenized to two spaces.
- spaces with character code different from space (32) after a control sequence is not ignored. i.e. in expl3 catcode, ~ always become a space.
- ^^ syntax are not supported. Use Python's escape syntax (e.g.) as usual (of course that does not work in raw Python strings `r"..."`).

Parameters
get_catcode – A function that given a character code, return its desired category code.

is_balanced() → bool

See *NTokenList.is_balanced()*.

static iterable_from_string(*s: str*, *get_catcode: Callable[[int],* Catcode*]*) → Iterable[*Token*]

refer to documentation of from_string() for details.

put_next(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Put this token list forward in the input stream.

serialize() → str

serialize_bytes(*engine:* Engine) → bytes

Internal function.

Given an engine, serialize it in a form that is suitable for writing directly to the engine.

simple_detokenize(*get_catcode: Callable[[int],* Catcode*]*) → str

str(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

See *NTokenList.str()*.

str_unicode() → str

See *NTokenList.str_unicode()*.

```
pythonimmediate.bootstrap_code_functions:  list[EngineDependentCode] = [<function
_naive_flush_data_define>, functools.partial(<function naive_replace>,
'\n\\cs_new_protected:Npn \\__read_do_one_command:
{\n\t\\begingroup\n\t\t\\endlinechar=-1~\n\t\t\\readline \\__read_file to
\\__line\n\t\t\\expandafter\n\t\\endgroup % also this will give an error instead of
silently do nothing when command is invalid\n\t\t\\csname __run_ \\__line
:\\endcsname\n}\n\n\n% read documentation of ``_peek`` commands for details what this
command does.\n\\cs_new_protected:Npn \\pythonimmediatecontinue #1
{\n\t\\__send_content:e {r #1 %naive_inline%
}\n\t\\__read_do_one_command:\n}\n\n\\cs_new_protected:Npn \\pythonimmediatecontinuenoarg
{\n\t\\pythonimmediatecontinue {}\n}\n\n\n\\cs_new_protected:Npn \\__send_content:n #1
{\n\t\\__send_content:e { \\unexpanded{#1} }\n}\n\n\\cs_new_protected:Npn
\\__send_content_naive_flush:e #1 {\n\t\\__send_content:e { #1 %naive_inline%
}\n}\n\n\\cs_new_protected:Npn \\__send_content_naive_flush:n #1
{\n\t\\__send_content_naive_flush:e { \\unexpanded{#1} }\n}\n\n% the names are such that
\\__send_content%naive_send%:n {something} results in the correct content\n\n% internal
function. Just send an arbitrary block of data to Python.\n% this function only works
properly when newlinechar = 10.\n\\cs_new_protected:Npn \\__send_block:e #1
{\n\t\\__send_content:e {\n\t\t#1 ^^J\n\t\tpythonimm?"""?\'\'\'? % following character
will be newline\n\t}\n}\n\n\\cs_new_protected:Npn \\__send_block:n #1
{\n\t\\__send_block:e {\\unexpanded{#1}}\n}\n\n\\cs_new_protected:Npn
\\__send_block_naive_flush:e #1 {\n\t\\__send_content:e {\n\t\t#1
^^J\n\t\tpythonimm?"""?\'\'\'? % following character will be
newline\n\t\t%naive_inline%\n\t}\n}\n\n\\cs_new_protected:Npn
\\__send_block_naive_flush:n #1 {\n\t\\__send_block_naive_flush:e
{\\unexpanded{#1}}\n}\n\n\\cs_generate_variant:Nn \\__send_block:n
{V}\n\\cs_generate_variant:Nn \\__send_block_naive_flush:n
{V}\n\n\\AtEndDocument{\n\t\\__send_content:e {r
%naive_inline%}\n\t\\__close_write:\n}\n'), <function mark_bootstrap.<locals>.<lambda>>,
functools.partial(<function naive_replace>, '\n\t\\cs_new_protected:Npn \\py#1
{\n\t\t\\__send_content:e { i a }\n\t\t\\__begingroup_setup_estr:
\\__send_block%naive_send%:e { #1 } \\endgroup\n\t\t\\__read_do_one_command:\n\t}\n\t'),
functools.partial(<function naive_replace>, '\n\t\\cs_new_protected:Npn \\pyfile#1
{\n\t\t\\__send_content:e { i b }\n\t\t\\__begingroup_setup_estr:
\\__send_content%naive_send%:e { #1 }
\\endgroup\n\t\t\\__read_do_one_command:\n\t}\n\t'), functools.partial(<function
naive_replace>, '\n\t\\cs_new_protected:Npn \\pyc#1 {\n\t\t\\__send_content:e { i c
}\n\t\t\\__begingroup_setup_estr:  \\__send_block%naive_send%:e { #1 }
\\endgroup\n\t\t\\__read_do_one_command:\n\t}\n\t'), functools.partial(<function
naive_replace>, '\n\t\\cs_new_protected:Npn \\pycq#1 {\n\t\t\\__send_content:e { i d
}\n\t\t\\__begingroup_setup_estr:  \\__send_block%naive_send%:e { #1 }
\\endgroup\n\t\t\\__read_do_one_command:\n\t}\n\t'), <function
mark_bootstrap.<locals>.<lambda>>, <function mark_bootstrap.<locals>.<lambda>>,
functools.partial(<function naive_replace>, '\n\t\\cs_new_protected:Npn
\\__pycodex#1#2#3#4 {\n\t\t\\__send_content:e { i e }\n\t\t\\__send_block:n { #1 }
\\__send_content:n { #2 }\\__send_content:n { #3 }\\__send_content%naive_send%:n { #4
}\n\t\t\\__read_do_one_command:\n\t}\n\t'), functools.partial(<function naive_replace>,
'\n\\cs_new_eq:NN \\__run_f:  \\relax\n'), functools.partial(<function naive_replace>,
'\n\\msg_new:nnn {pythonimmediate} {python-error}
{Python~error:~#1.}\n\\cs_new_protected:Npn \\__run_g:  {\n\t\\__read_block:N
\\__data\n\t\\__read_block:N
\\__summary\n\t\\wlog{^^JPython~error~traceback:^^J\\__data^^J}\n \\msg_error:nnx
{pythonimmediate} {python-error} {\\__summary}\n}\n'), functools.partial(<function
naive_replace>, '\n\t\t\\cs_new_protected:Npn \\__run_h:  {\n\t\t\t\\__read_block:N
\\__data\n\t\t\t\\begingroup \\newlinechar=10~ \\expandafter
\\endgroup\n\t\t\t\\scantokens \\expandafter{\\__data}\n\t\t\t% trick described in
https://tex.stackexchange.com/q/640274 to scantokens the code with
\\newlinechar=10\n\n\t\t\t\\__send_content%naive_send%:e { r
}\n\t\t\t\\__read_do_one_command:\n\t\t}\n\t\t'), functools.partial(<function
naive_replace>, '\n\t\t\\cs_new_eq:NN \\__run_i:  \\relax\n\t\t'),
```

Internal constant. Contains functions that takes an engine object and returns some code before *substitute_private()* is applied on it.

pythonimmediate.**build_Python_call_TeX**(*T: Type*, *TeX_code: str*, *, *recursive: bool = True*, *sync: Optional[bool] = None*, *finish: bool = False*) → None

Internal function. See *scan_Python_call_TeX()*.

T has the form Callable[[T1, T2], Tuple[U1, U2]] where the Tx are subclasses of PyToTeXData and the Ux are subclasses of TeXToPyData

The Tuple[...] can optionally be a single type, then it is almost equivalent to a tuple of one element It can also be None

pythonimmediate.**can_be_mangled_to**(*original: str*, *mangled: str*) → bool

If *original* is put in a TEX file, read in other catcode regime (possibly drop trailing spaces/tabs), and then sent through `\write` (possibly convert control characters to `^^`-notation), is it possible that the written content is equal to *mangled*?

The function is somewhat tolerant (might return `True` in some cases where `False` should be returned), but not too tolerant.

Example:

```
>>> can_be_mangled_to("a\n", "a\n")
True
>>> can_be_mangled_to("\n", "\n")
True
>>> can_be_mangled_to("\t\n", "\n")
True
>>> can_be_mangled_to("\t\n", "\t\n")
True
>>> can_be_mangled_to("\t\n", "^^I\n")
True
>>> can_be_mangled_to("\ta\n", "^^Ia\n")
True
>>> can_be_mangled_to("a b\n", "a b\n")
True
>>> can_be_mangled_to("a b  \n", "a b\n")
True
>>> can_be_mangled_to("a\n", "b\n")
False
```

pythonimmediate.**check_line**(*line: str*, *, *braces: bool*, *newline: bool*, *continue_: Optional[bool]*) → None

check user-provided line before sending to TeX for execution

pythonimmediate.**continue_until_passed_back**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Same as `continue_until_passed_back_str()` but nothing can be returned from TeX to Python.

pythonimmediate.**continue_until_passed_back_str**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Usage:

First put some tokens in the input stream that includes `\pythonimmediatecontinue{...}` (or `%sync% \__read_do_one_command:`), then call `continue_until_passed_back()`.

The function will only return when the `\pythonimmediatecontinue` is called.

pythonimmediate.**debug**(**args*, ***kwargs*)

pythonimmediate.**define_Python_call_TeX**(*TeX_code: str*, *ptt_argtypes: List[Type[PyToTeXData]]*, *ttp_argtypes: List[Type[TeXToPyData]]*, ***, *recursive: bool = True*, *sync: Optional[bool] = None*, *finish: bool = False*) → Tuple[Callable[[*Engine*], str], *PythonCallTeXFunctionType*]

Internal function.

`TeX_code` should be some expl3 code that defines a function with name `%name%` that when called should:

- run some $\TeX$-code (which includes reading the arguments, if any)
- **do the following if `sync`:**
  - send `r` to Python (equivalently write %sync%)
  - send whatever needed for the output (as in `ttp_argtypes`)
- call `\__read_do_one_command:` iff not `finish`.

This is allowed to contain the following: * %name%: the name of the function to be defined as explained above. * %read_arg0(var_name)%, %read_arg1(...)%: will be expanded to code that reads the input. * %send_arg0(...)%, %send_arg1(...)%: will be expanded to code that sends the content. * %send_arg0_var(var_name)%, %send_arg1_var(...)%: will be expanded to code that sends the content in the variable. * %optional_sync%: expanded to code that writes `r` (to sync), if `sync` is True. * %naive_flush% and %naive_inline%: as explained in *mark_bootstrap_naive_replace()*.

(although usually you don't need to explicitly write this, it's embedded in the `send*()` command of the last argument, or `%sync%`)

ptt_argtypes: list of argument types to be sent from Python to TeX (i.e. input of the TeX function)

ttp_argtypes: list of argument types to be sent from TeX to Python (i.e. output of the TeX function)

recursive: whether the TeX_code might call another Python function. Default to True.
It does not hurt to always specify True, but performance would be a bit slower.

sync: whether the Python function need to wait for the TeX function to finish.
Required if `ttp_argtypes` is not empty. This should be left to be the default None most of the time. (which will make it always sync if `debugging`, otherwise only sync if needed i.e. there's some output)

finish: Include this if and only if `\__read_do_one_command:` is omitted.
Normally this is not needed, but it can be used as a slight optimization; and it's needed internally to implement `run_none_finish` among others. For each TeX-call-Python layer, emph{exactly one} `finish` call can be made. If the function itself doesn't call any `finish` call (which happens most of the time), then the wrapper will call `run_none_finish`.

Return some TeX code to be executed, and a Python function object that when called will call the TeX function on the passed-in TeX engine and return the result.

Note that the TeX_code must eventually be executed on the corresponding engine for the program to work correctly.

Possible optimizations:

- **the `r` is not needed if not recursive and `ttp_argtypes` is nonempty**
  (the output itself tells Python when the $\TeX$-code finished)
- the first line of the output may be on the same line as the `r` itself (done, use *TTPEmbeddedLine* type, although a bit hacky)

pythonimmediate.**define_TeX_call_Python**(*f: Callable[[...], None]*, *name: Optional[str] = None*, *argtypes: Optional[List[Type[TeXToPyData]]] = None*, *identifier: Optional[str] = None*) → Callable[[*Engine*], str]

This function setups some internal data structure, and returns the TEX-code to be executed on the TEX-side to define the macro.

f: the Python function to be executed. It should take some arguments plus a keyword argument *engine* and eventually (optionally) call one of the `_finish` functions.

name: the macro name on the TEX-side. This should only consist of letter characters in `expl3` catcode regime.

argtypes: list of argument types. If it's None it will be automatically deduced from the function `f`'s signature.

Returns: some code (to be executed in `expl3` catcode regime) as explained above.

pythonimmediate.**define_internal_handler**(*f: Callable*) → Callable

define a TeX function with TeX name = f.__name__ that calls f().

this does not define the specified function in any particular engine, just add them to the bootstrap_code.
essert self.process is not None, "process is already closed!"

pythonimmediate.**eval_with_linecache**(*code: str*, *globals: Dict[str, Any]*) → Any

pythonimmediate.**exec_or_eval_with_linecache**(*code: str*, *globals: dict*, *mode: str*) → Any

pythonimmediate.**exec_with_linecache**(*code: str*, *globals: Dict[str, Any]*) → None

pythonimmediate.**expand_once**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

Expand the following content in the input stream once.

For example, if the following tokens in the input stream are `\iffalse 1 \else 2 \fi`, then after `expand_once()` being called once, the tokens in the input stream will be `2 \fi`.

pythonimmediate.**export_function_to_module**(*f: FunctionType*) → FunctionType

the functions decorated with this decorator are accessible from user code with

import pythonimmediate pythonimmediate.function name(...)

pythonimmediate.**get_bootstrap_code**(*engine: Engine*) → str

Return the bootstrap code for an engine.

This is before the call to *substitute_private()*.

pythonimmediate.**get_random_identifier**() → str

pythonimmediate.**have_naive_replace**(*code: str*) → bool

pythonimmediate.**mark_bootstrap**(*code: str*) → None

pythonimmediate.**mark_bootstrap_naive_replace**(*code: str*) → None

Similar to *mark_bootstrap()*, but code may contain one of the following:

- **%naive_inline: replaced with ^^J __naive_flush_data:**
  if `Engine.config.naive_flush` is `True`, else become empty
- **%naive_flush%: replaced with __send_content:e {__naive_flush_data:}**
  if `Engine.config.naive_flush` is `True`

pythonimmediate.**naive_replace**(*code: str*, *naive_flush: bool*, */*) → str

pythonimmediate.**naive_replace**(*code: str*, *engine: Engine*, */*) → str

pythonimmediate.**normalize_line**(*line: str*) → str

pythonimmediate.**parse_meaning_str**(*s: str*) → Optional[Tuple[*Catcode*, str]]

pythonimmediate.**peek_next_meaning**(*engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

Get the meaning of the following token, as a string, using the current `\escapechar`.

This is recommended over `peek_next_token()` as it will not tokenize an extra token.

It's undefined behavior if there's a newline (`\newlinechar` or `^^J`, the latter is OS-specific) in the meaning string.

pythonimmediate.**postprocess_send_code**(*s: str*, *put_sync: bool*) → str

pythonimmediate.**py**(*code:* TTPEBlock, *engine:* Engine) → None

pythonimmediate.**pyc**(*code:* TTPEBlock, *engine:* Engine) → None

pythonimmediate.**pycq**(*code:* TTPEBlock, *engine:* Engine) → None

pythonimmediate.**pyfile**(*filename:* TTPELine, *engine:* Engine) → None

pythonimmediate.**random_identifiers**() → Iterator[str]

pythonimmediate.**read_block**(*engine:* Engine) → str

Internal function to read one block sent from :math:`TeX` (including the final delimiter line, but the delimiter line is not returned)

pythonimmediate.**readline**(*engine:* Engine) → str

pythonimmediate.**run_block_finish**(*block: str*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

pythonimmediate.**run_block_local**(*block: str*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → None

pythonimmediate.**run_code_redirect_print_TeX**(*f: Callable[[], Any]*, *engine:* Engine) → None

pythonimmediate.**run_error_finish**(**args*)

pythonimmediate.**run_main_loop**(*engine:* Engine) → Optional[str]

pythonimmediate.**run_main_loop_get_return_one**(*engine:* Engine) → str

pythonimmediate.**run_none_finish**(**args*)

Internal function.

`run_error_finish` is fatal to TEX, so we only run it when it's fatal to Python.

We want to make sure the Python traceback is printed strictly before run_error_finish() is called, so that the Python traceback is not interleaved with TEX error messages.

pythonimmediate.**run_tokenized_line_peek**(*line: str*, ***, *check_braces: bool = True*, *check_newline: bool = True*, *check_continue: bool = True*, *engine: ~pythonimmediate.engine.Engine = <pythonimmediate.engine.DefaultEngine object>*) → str

pythonimmediate.**scan_Python_call_TeX**(*sourcecode: str*) → None

Internal function.

Scan the file in filename for occurrences of `typing.cast(T, Python_call_TeX_local(...))`, then call `build_Python_call_TeX(T, ...)` for each occurrence.

The way the whole thing work is:

- In the Python code, some `typing.cast(T, Python_call_TeX_local(...))` are used.
- This function is called on all the library source codes to scan for those occurrences, build necessary data structures for the *Python_call_TeX_local()* function calls to work correctly.
- When *Python_call_TeX_local()* is actually called, it does some magic to return the correct function.

Done this way, the type checking works correctly and it's not necessary to define global temporary variables.

Don't use this function on untrusted code.

pythonimmediate.**scan_Python_call_TeX_module**(*name: str*) → None

Internal function. Can be used as `scan_Python_call_TeX_module(__name__)` to scan the current module.

pythonimmediate.**send_bootstrap_code**(*engine:* Engine) → None

pythonimmediate.**send_finish**(*s: str*, *engine:* Engine) → None

pythonimmediate.**send_raw**(*s: str*, *engine:* Engine) → None

pythonimmediate.**substitute_private**(*code: str*) → str

pythonimmediate.**surround_delimiter**(*block: str*) → str

pythonimmediate.**template_substitute**(*template: str*, *pattern: str*, *substitute: Union[str, Callable[[Match], str]]*, *optional: bool = False*) → str

pattern is a regex

pythonimmediate.**user_documentation**(*x: T1*) → T1

pythonimmediate.**wrap_naive_replace**(*code: str*) → Callable[[*Engine*], str]

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W