bpc-utils

Release 0.10.1

Python Backport Compiler Project

Apr 02, 2021

CONTENTS

Utility library for the Python `bpc` backport compiler.

Currently, the three individual tools (`f2format`, `poseur`, `walrus`) depend on this repo. The `bpc` compiler is a work in progress.

CHAPTER

ONE

MODULE CONTENTS

Utility library for the Python bpc backport compiler.

exception bpc_utils.**BPCInternalError**(*message*, *context*)

Bases: RuntimeError

Internal bug happened in BPC tools.

Initialize BPCInternalError.

Parameters

- **message** (object) – the error message
- **context** (str) – describe the context/location/component where the bug happened

Raises

- **TypeError** – if context is not str
- **ValueError** – if message (when converted to str) or context is empty or only contains whitespace characters

exception bpc_utils.**BPCRecoveryError**

Bases: RuntimeError

Error during file recovery.

exception bpc_utils.**BPCSyntaxError**

Bases: SyntaxError

Syntax error detected when parsing code.

class bpc_utils.**BaseContext**(*node*, *config*, *, *indent_level=0*, *raw=False*)

Bases: abc.ABC

Abstract base class for general conversion context.

Initialize BaseContext.

Parameters

- **node** (*parso.tree.NodeOrLeaf*) – parso AST
- **config** (*Config*) – conversion configurations
- **indent_level** (int) – current indentation level
- **raw** (bool) – raw processing flag

final **__iadd__**(*code*)

Support of the += operator.

If *self._prefix_or_suffix* is `True`, then the `code` will be appended to *self._prefix*; else it will be appended to *self._suffix*.

Parameters **code** (`str`) – code string

Returns self

Return type *BaseContext*

final __str__()
Returns a *stripped* version of *self._buffer*.

Return type `str`

abstract _concat()
Concatenate final string.

Return type `None`

final _process(*node*)
Recursively process parso AST.

All processing methods for a specific `node` type are defined as `_process_{type}`. This method first checks if such processing method exists. If so, it will call such method on the `node`; otherwise it will traverse through all *children* of `node`, and perform the same logic on each child.

Parameters **node** (*parso.tree.NodeOrLeaf*) – parso AST

Return type `None`

final _walk(*node*)
Start traversing the AST module.

The method traverses through all *children* of `node`. It first checks if such child has the target expression. If so, it will toggle *self._prefix_or_suffix* (set to `False`) and save the last previous child as *self._node_before_expr*. Then it processes the child with *self._process*.

Parameters **node** (*parso.tree.NodeOrLeaf*) – parso AST

Return type `None`

final static extract_whitespaces(*code*)
Extract preceding and succeeding whitespaces from the code given.

Parameters **code** (`str`) – the code to extract whitespaces

Return type Tuple[str, str]

Returns a tuple of *preceding* and *succeeding* whitespaces in `code`

abstract has_expr(*node*)
Check if node has the target expression.

Parameters **node** (*parso.tree.NodeOrLeaf*) – parso AST

Return type `bool`

Returns whether `node` has the target expression

final classmethod mangle(*cls_name*, *var_name*)
Mangle variable names.

This method mangles variable names as described in Python documentation about mangling and further normalizes the mangled variable name through *normalize()*.

Parameters

- **cls_name** (str) – class name
- **var_name** (str) – variable name

Return type str

Returns mangled and normalized variable name

final static missing_newlines(*prefix*, *suffix*, *expected*, *linesep*)
Count missing blank lines for code insertion given surrounding code.

Parameters

- **prefix** (str) – preceding source code
- **suffix** (str) – succeeding source code
- **expected** (int) – number of expected blank lines
- **linesep** (*Linesep*) – line seperator

Return type int

Returns number of blank lines to add

final static normalize(*name*)
Normalize variable names.

This method normalizes variable names as described in Python documentation about identifiers and **PEP 3131**.

Parameters **name** (str) – variable name as it appears in the source code

Return type str

Returns normalized variable name

final static split_comments(*code*, *linesep*)
Separates prefixing comments from code.

This method separates *prefixing* comments and *suffixing* code. It is rather useful when inserting code might break shebang and encoding cookies (**PEP 263**), etc.

Parameters

- **code** (str) – the code to split comments
- **linesep** (*Linesep*) – line seperator

Return type Tuple[str, str]

Returns a tuple of *prefix comments* and *suffix code*

_buffer: str
Final converted result.

_indent_level: Final[int]
Current indentation level.

_indentation: Final[str]
Indentation sequence.

_linesep: Final[Linesep]
Line seperator.

Type Final[*Linesep*]

_node_before_expr: Optional[parso.tree.NodeOrLeaf]
Preceding node with the target expression, i.e. the *insertion point*.

_pep8: Final[bool]
PEP 8 compliant conversion flag.

_prefix: str
Code before insertion point.

_prefix_or_suffix: bool
Flag to indicate whether buffer is now *self._prefix*.

_root: Final[parso.tree.NodeOrLeaf]
Root node given by the `node` parameter.

_suffix: str
Code after insertion point.

_uuid_gen: Final[*UUID4Generator*]
UUID generator.

config: Final[*Config*]
Internal configurations.

property string
Returns conversion buffer (*self._buffer*).

Return type str

class bpc_utils.**Config**(***kwargs*)
Bases: MutableMapping[str, object]

Configuration namespace.

This class is inspired from `argparse.Namespace` for storing internal attributes and/or configuration variables.

```
>>> config = Config(foo='var', bar=True)
>>> config.foo
'var'
>>> config['bar']
True
>>> config.bar = 'boo'
>>> del config['foo']
>>> config
Config(bar='boo')
```

class bpc_utils.**Placeholder**(*name*)
Bases: object

Placeholder for string interpolation.

Placeholder objects can be concatenated with `str`, other *Placeholder* objects and *StringInterpolation* objects via the '+' operator.

Placeholder objects should be regarded as immutable. Please do not modify the `name` attribute. Build new objects instead.

Initialize Placeholder.

Parameters **name** (str) – name of the placeholder

Raises **TypeError** – if `name` is not `str`

class bpc_utils.**StringInterpolation**(**args*)

Bases: object

A string with placeholders to be filled in.

This looks like an object-oriented format string, but making sure that string literals are always interpreted literally (so no need to manually do escaping). The boundaries between string literals and placeholders are very clear. Filling in a placeholder will never inject a new placeholder, protecting string integrity for multiple-round interpolation.

```
>>> s1 = '%(injected)s'
>>> s2 = 'hello'
>>> s = StringInterpolation('prefix ', Placeholder('q1'), ' infix ', Placeholder(
↪'q2'), ' suffix')
>>> str(s % {'q1': s1} % {'q2': s2})
'prefix %(injected)s infix hello suffix'
```

(This can be regarded as an improved version of `string.Template.safe_substitute()`.)

Multiple-round interpolation is tricky to do with a traditional format string. In order to do things correctly and avoid format string injection vulnerabilities, you need to perform escapes very carefully.

```
>>> fs = 'prefix %(q1)s infix %(q2)s suffix'
>>> fs % {'q1': s1} % {'q2': s2}
Traceback (most recent call last):
    ...
KeyError: 'q2'
>>> fs = 'prefix %(q1)s infix %%(q2)s suffix'
>>> fs % {'q1': s1} % {'q2': s2}
Traceback (most recent call last):
    ...
KeyError: 'injected'
>>> fs % {'q1': s1.replace('%', '%%')} % {'q2': s2}
'prefix %(injected)s infix hello suffix'
```

StringInterpolation objects can be concatenated with `str`, *Placeholder* objects and other *StringInterpolation* objects via the '+' operator.

StringInterpolation objects should be regarded as immutable. Please do not modify the `literals` and `placeholders` attributes. Build new objects instead.

Initialize StringInterpolation. `args` will be concatenated to construct a *StringInterpolation* object.

```
>>> StringInterpolation('prefix', Placeholder('data'), 'suffix')
StringInterpolation('prefix', Placeholder('data'), 'suffix')
```

Parameters **args** (*Union[str, Placeholder, StringInterpolation]*) – the components to construct a *StringInterpolation* object

__mod__(*substitutions*)

Substitute the placeholders in this *StringInterpolation* object with string values (if possible) according to the `substitutions` mapping.

```
>>> StringInterpolation('prefix ', Placeholder('data'), ' suffix') % {'data':
↪'hello'}
StringInterpolation('prefix hello suffix')
```

Parameters **substitutions** (*Mapping[str, object]*) – a mapping from placeholder names to the values to be filled in; all values are converted into `str`

Return type *StringInterpolation*

Returns a new *StringInterpolation* object with as many placeholders substituted as possible

__str__()

Returns the fully-substituted string interpolation result.

```
>>> str(StringInterpolation('prefix hello suffix'))
'prefix hello suffix'
```

Return type `str`

Returns the fully-substituted string interpolation result

Raises **ValueError** – if there are still unsubstituted placeholders in this *StringInterpolation* object

static from_components(*literals*, *placeholders*)

Construct a *StringInterpolation* object from `literals` and `placeholders` components. This method is more efficient than the *StringInterpolation()* constructor, but it is mainly intended for internal use.

```
>>> StringInterpolation.from_components(
...     ('prefix', 'infix', 'suffix'),
...     (Placeholder('data1'), Placeholder('data2'))
... )
StringInterpolation('prefix', Placeholder('data1'), 'infix', Placeholder(
↪'data2'), 'suffix')
```

Parameters

- **literals** (*Iterable[str]*) – the literal components in order
- **placeholders** (*Iterable[Placeholder]*) – the *Placeholder* components in order

Return type *StringInterpolation*

Returns the constructed *StringInterpolation* object

Raises

- **TypeError** – if `literals` is `str`; if `literals` contains non-`str` values; if `placeholders` contains non-*Placeholder* values
- **ValueError** – if the length of `literals` is not exactly one more than the length of `placeholders`

iter_components()

Generator to iterate all components of this *StringInterpolation* object in order.

```
>>> list(StringInterpolation('prefix', Placeholder('data'), 'suffix').iter_
↪components())
['prefix', Placeholder('data'), 'suffix']
```

Return type Generator[Union[str, *Placeholder*], None, None]

Returns generator containing the components of this *StringInterpolation* object in order

property result

Alias of *StringInterpolation.__str__()* to get the fully-substituted string interpolation result.

```
>>> StringInterpolation('prefix hello suffix').result
'prefix hello suffix'
```

Return type str

class bpc_utils.**UUID4Generator**(*dash=True*)

Bases: object

UUID 4 generator wrapper to prevent UUID collisions.

Constructor of UUID 4 generator wrapper.

Parameters **dash** (bool) – whether the generated UUID string has dashes or not

gen()

Generate a new UUID 4 string that is guaranteed not to collide with used UUIDs.

Return type str

Returns a new UUID 4 string

bpc_utils.**TaskLock**()

Function that returns a lock for possibly concurrent tasks.

Return type ContextManager[None]

Returns a lock for possibly concurrent tasks

bpc_utils.**archive_files**(*files*, *archive_dir*)

Archive the list of files into a *tar* file.

Parameters

- **files** (*Iterable[str]*) – a list of files to be archived (should be *absolute path*)
- **archive_dir** (str) – the directory to save the archive

Return type str

Returns path to the generated *tar* archive

bpc_utils.**detect_encoding**(*code*)

Detect encoding of Python source code as specified in **PEP 263**.

Parameters **code** (bytes) – the code to detect encoding

Return type str

Returns the detected encoding, or the default encoding (utf-8)

Raises

- **TypeError** – if code is not a bytes string
- **SyntaxError** – if both a BOM and a cookie are present, but disagree

bpc_utils.**detect_files**(*files*)

Get a list of Python files to be processed according to user input.

This will perform *glob* expansion on Windows, make all paths absolute, resolve symbolic links and remove duplicates.

Parameters **files** (*Iterable[str]*) – a list of files and directories to process (usually provided by users on command-line)

Return type List[str]

Returns a list of Python files to be processed

See also:

See *expand_glob_iter()* for more information.

bpc_utils.**detect_indentation**(*code*)

Detect indentation of Python source code.

Parameters **code** (*Union[str, bytes, TextIO, parso.tree.NodeOrLeaf]*) – the code to detect indentation

Return type str

Returns the detected indentation sequence

Raises **TokenError** – when failed to tokenize the source code under certain cases, see documentation of TokenError for more details

---

Notes

In case of mixed indentation, try voting by the number of occurrences of each indentation value (*spaces* and *tabs*).

When there is a tie between *spaces* and *tabs*, prefer **4 spaces** for **PEP 8**.

---

bpc_utils.**detect_linesep**(*code*)

Detect linesep of Python source code.

Parameters **code** (*Union[str, bytes, TextIO, parso.tree.NodeOrLeaf]*) – the code to detect linesep

Returns the detected linesep (one of '\n', '\r\n' and '\r')

Return type *Linesep*

Notes

In case of mixed linesep, try voting by the number of occurrences of each linesep value.

When there is a tie, prefer LF to CRLF, prefer CRLF to CR.

---

bpc_utils.**first_non_none**(**args*)

Return the first non-None value from a list of values.

Parameters ***args** – variable length argument list

- If one positional argument is provided, it should be an iterable of the values.
- If two or more positional arguments are provided, then the value list is the positional argument list.

Returns the first non-`None` value, if all values are `None` or sequence is empty, return `None`

Raises **`TypeError`** – if no arguments provided

bpc_utils.**first_truthy**(**args*)
Return the first *truthy* value from a list of values.

Parameters **`*args`** – variable length argument list

- If one positional argument is provided, it should be an iterable of the values.
- If two or more positional arguments are provided, then the value list is the positional argument list.

Returns the first *truthy* value, if no *truthy* values found or sequence is empty, return `None`

Raises **`TypeError`** – if no arguments provided

bpc_utils.**getLogger**(*name*, *level=20*)
Create a BPC logger.

Parameters

- **`name`** (`str`) – name for the logger
- **`level`** (`int`) – log level for the logger

Return type `Logger`

Returns the created logger

bpc_utils.**get_parso_grammar_versions**(*minimum=None*)
Get Python versions that parso supports to parse grammar.

Parameters **`minimum`** (*`Optional[str]`*) – filter result by this minimum version

Return type List[str]

Returns a list of Python versions that parso supports to parse grammar

Raises

- **`TypeError`** – if `minimum` is not `str`
- **`ValueError`** – if `minimum` is invalid

bpc_utils.**map_tasks**(*func*, *iterable*, *posargs=None*, *kwargs=None*, ***, *processes=None*, *chunksize=None*)
Execute tasks in parallel if `multiprocessing` is available, otherwise execute them sequentially.

Parameters

- **`func`** (*`Callable[.., T]`*) – the task function to execute
- **`iterable`** (*`Iterable[object]`*) – the items to process
- **`posargs`** (*`Optional[Iterable[object]]`*) – additional positional arguments to pass to `func`
- **`kwargs`** (*`Optional[Mapping[str, object]]`*) – keyword arguments to pass to `func`
- **`processes`** (*`Optional[int]`*) – the number of worker processes (default: auto determine)
- **`chunksize`** (*`Optional[int]`*) – chunk size for multiprocessing

Return type List[T]

Returns the return values of the task function applied on the input items and additional arguments

bpc_utils.**parse_boolean_state**(*s*)

Parse a boolean state from a string representation.

- These values are regarded as `True`: `'1'`, `'yes'`, `'y'`, `'true'`, `'on'`
- These values are regarded as `False`: `'0'`, `'no'`, `'n'`, `'false'`, `'off'`

Value matching is case **insensitive**.

Parameters **s** (*Optional[str]*) – string representation of a boolean state

Return type Optional[bool]

Returns the parsed boolean result, return `None` if input is `None`

Raises **ValueError** – if `s` is an invalid boolean state value

See also:

See *_boolean_state_lookup* for default lookup mapping values.

bpc_utils.**parse_indentation**(*s*)

Parse indentation from a string representation.

- If an integer or a string of positive integer `n` is specified, then indentation is `n` spaces.
- If `'t'` or `'tab'` is specified, then indentation is tab.
- If `'\t'` (the tab character itself) or a string consisting only of the space character (U+0020) is specified, it is returned directly.

Value matching is **case insensitive**.

Parameters **s** (*Optional[Union[str, int]]*) – string representation of indentation

Return type Optional[str]

Returns the parsed indentation result, return `None` if input is `None` or empty string

Raises

- **TypeError** – if `s` is not `str` or `int`
- **ValueError** – if `s` is an invalid indentation value

bpc_utils.**parse_linesep**(*s*)

Parse linesep from a string representation.

- These values are regarded as `'\n'`: `'\n'`, `'lf'`
- These values are regarded as `'\r\n'`: `'\r\n'`, `'crlf'`
- These values are regarded as `'\r'`: `'\r'`, `'cr'`

Value matching is **case insensitive**.

Parameters **s** (*Optional[str]*) – string representation of linesep

Returns the parsed linesep result, return `None` if input is `None` or empty string

Return type Optional[*Linesep*]

Raises **ValueError** – if `s` is an invalid linesep value

See also:

See *_linesep_lookup* for default lookup mapping values.

bpc_utils.**parse_positive_integer**(*s*)
Parse a positive integer from a string representation.

Parameters **s** (*Optional[Union[str, int]]*) – string representation of a positive integer, or just an integer

Return type Optional[int]

Returns the parsed integer result, return None if input is None or empty string

Raises

- **TypeError** – if s is not str or int
- **ValueError** – if s is an invalid positive integer value

bpc_utils.**parso_parse**(*code*, *filename=None*, *, *version=None*)
Parse Python source code with parso.

Parameters

- **code** (*Union[str, bytes]*) – the code to be parsed
- **filename** (*Optional[str]*) – an optional source file name to provide a context in case of error
- **version** (*Optional[str]*) – parse the code as this version (uses the latest version by default)

Return type parso.python.tree.Module

Returns parso AST

Raises ***BPCSyntaxError*** – when source code contains syntax errors

bpc_utils.**recover_files**(*archive_file_or_dir*, *, *rr=False*, *rs=False*)
Recover files from a *tar* archive, optionally removing the archive file and archive directory after recovery.

This function supports three modes:

- **Normal mode (when rr and rs are both False):** Recover from the archive file specified by archive_file_or_dir.
- **Recover and remove (when rr is True):** Recover from the archive file specified by archive_file_or_dir, and remove this archive file after recovery.
- **Recover from the only file in the archive directory (when rs is True):** If the directory specified by archive_file_or_dir contains exactly one (regular) file, recover from that file and remove the archive directory.

Specifying both rr and rs as True is not accepted.

Parameters

- **archive_file** – path to the *tar* archive file, or the archive directory
- **rr** (bool) – whether to run in "recover and remove" mode
- **rs** (bool) – whether to run in "recover from the only file in the archive directory" mode

Raises

- **ValueError** – when rr and rs are both True
- ***BPCRecoveryError*** – when rs is True, and the directory specified by archive_file_or_dir is empty, contains more than one item, or contains a non-regular file

Return type None

bpc_utils.**Linesep**
Type alias for `Literal['\n', '\r\n', '\r']`.

CHAPTER

TWO

INTERNAL UTILITIES

bpc_utils.argparse.**_boolean_state_lookup**

Type Final[Dict[str, bool]]

A mapping from string representation to boolean states. The values are used for *parse_boolean_state()*.

bpc_utils.argparse.**_linesep_lookup**

Type Final[Dict[str, *Linesep*]]

A mapping from string representation to linesep. The values are used for *parse_linesep()*.

bpc_utils.fileprocessing.**has_gz_support**

Type bool

Whether gzip is supported.

bpc_utils.fileprocessing.**LOOKUP_TABLE: Final[str] = '_lookup_table.json'**
File name for the lookup table in the archive file.

Type Final[str]

bpc_utils.fileprocessing.**is_python_filename**(*filename*)
Determine whether a file is a Python source file by its extension.

Parameters **filename** (str) – the name of the file

Return type bool

Returns whether the file is a Python source file

bpc_utils.fileprocessing.**expand_glob_iter**(*pattern*)
Wrapper function to perform glob expansion.

Parameters **pattern** (str) – the pattern to expand

Return type Iterator[str]

Returns an iterator of expansion result

class bpc_utils.logging.**BPCLogHandler**
Bases: logging.StreamHandler

Handler used to format BPC logging records.

Initialize BPCLogHandler.

format(*record*)
Format the specified record based on log level.

The record will be formatted based on its log level in the following flavour:

DEBUG	[%(levelname)s] %(asctime)s %(message)s
INFO	%(message)s
WARNING	Warning: %(message)s
ERROR	Error: %(message)s
CRITICAL	Error: %(message)s

Parameters **record** (LogRecord) – the log record

Return type str

Returns the formatted log string

format_templates: Dict[str, str] = {'CRITICAL': 'Error: %(message)s', 'DEBUG': '[%(l

time_format = '%Y-%m-%d %H:%M:%S.%f%z'

bpc_utils.misc.**is_windows**

Type bool

Whether the current operating system is Windows.

bpc_utils.misc.**current_time_with_tzinfo**()
Get the current time with local time zone information.

Return type datetime

Returns datetime object representing current time with local time zone information

class bpc_utils.misc.**MakeTextIO**(*obj*)
Bases: object

Context wrapper class to handle str and *file* objects together.

Variables

- ***obj*** (*Union[str, TextIO]*) – the object to manage in the context
- ***sio*** (*Optional[StringIO]*) – the I/O object to manage in the context only if *self.obj* is str
- ***pos*** (*Optional[int]*) – the original offset of *self.obj*, only if *self.obj* is a seekable *file* object

Initialize context.

Parameters **obj** (*Union[str, TextIO]*) – the object to manage in the context

obj

Type Union[str, TextIO]

The object to manage in the context.

sio

Type StringIO

The I/O object to manage in the context only if *self.obj* is str.

pos

Type int

The original offset of *self.obj*, if only *self.obj* is a seekable TextIO.

__enter__()
Enter context.

- If *self.obj* is str, a StringIO will be created and returned.
- If *self.obj* is a seekable *file* object, it will be seeked to the beginning and returned.
- If *self.obj* is an unseekable *file* object, it will be returned directly.

Return type TextIO

__exit__(*exc_type*, *exc_value*, *traceback*)
Exit context.

- If *self.obj* is str, the StringIO (*self.sio*) will be closed.
- If *self.obj* is a seekable *file* object, its stream position (*self.pos*) will be recovered.

Return type None

bpc_utils.multiprocessing.**CPU_CNT**

Type int

Number of CPUs for multiprocessing support.

bpc_utils.multiprocessing.**mp**

Type Optional[ModuleType]

Value <module 'multiprocessing'>

An alias of the Python builtin multiprocessing module if available.

bpc_utils.multiprocessing.**parallel_available**

Type bool

Whether parallel execution is available.

bpc_utils.multiprocessing.**_mp_map_wrapper**(*args*)
Map wrapper function for multiprocessing.

Parameters **args** (*Tuple[Callable[.., T], Iterable[object], Mapping[str, object]]*) – the function to execute, the positional arguments and the keyword arguments packed into a tuple

Return type T

Returns the function execution result

bpc_utils.multiprocessing.**_mp_init_lock**(*lock*)
Initialize lock for multiprocessing.

Parameters **lock** (*ContextManager[None]*) – the lock to be shared among tasks

Return type None

bpc_utils.multiprocessing.**task_lock**

Type ContextManager[None]

A lock for possibly concurrent tasks.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F