bpc-utils

Release 0.8.0

Python Backport Compiler Project

Sep 13, 2020

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

__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*

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

Return type str

abstract _concat()
Concatenate final string.

Return type None

_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** (`NodeOrLeaf`) – parso AST

Return type `None`

_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** (`NodeOrLeaf`) – parso AST

Return type `None`

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

Parameters **node** (`NodeOrLeaf`) – parso AST

Return type `Tuple`[`str`, `str`]

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

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

Parameters **node** (`NodeOrLeaf`) – parso AST

Return type `bool`

Returns whether `node` has the target expression

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

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
Final converted result.

Type str

_indent_level
Current indentation level.

Type int

_indentation: str
Indentation sequence.

Type str

_linesep: Linesep
Line seperator.

Type *Linesep*

_node_before_expr
Preceding node with the target expression, i.e. the *insertion point*.

Type Optional[parso.tree.NodeOrLeaf]

_pep8: bool
PEP 8 compliant conversion flag.

Type bool

_prefix
Code before insertion point.

Type str

_prefix_or_suffix
Flag if buffer is now *self._prefix*.

Type bool

_root
Root node given by the `node` parameter.

Type parso.tree.NodeOrLeaf

_suffix
Code after insertion point.

Type str

_uuid_gen
UUID generator.

Type *UUID4Generator*

config
Internal configurations.

Type *Config*

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

Return type str

class bpc_utils.**Config**(***kwargs*)
Bases: collections.abc.MutableMapping

Configuration namespace.

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

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 AbstractContextManager[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

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, NodeOrLeaf]) – the code to detect indentation

Return type str

Returns the detected indentation sequence

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, 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.**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** (Union[str, int, None]) – 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** (Union[str, int, None]) – 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 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

bpc_utils.misc.**is_windows**

Type bool

Whether the current operating system is Windows.

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** (AbstractContextManager[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

G

H

I

L

M

O

P

R

S

T

U