pidgy Documentation

Release stable

Nov 27, 2021

Contents

pidgy is a literate programming designed specifically for jupyter computational notebooks. In pidgy, authors weave narrative and tangle code using markdown as a document formatting language. This implementation is designed specifically for the IPython.InteractiveShell.

pidgy literate programs are serialized in the notebook schema therefore they can be converted a variety of document types. The pidgy package itself demonstrates an literate programming approach to writing readable, reusable, and reproducible scientific literature in notebooks.

Literate programming and computing

pidgin is inspired by web and literate coffeescript.

{#{appendix.exports(intro)}#}

CHAPTER 1

The `pidgy` implementation

`pidgy` written as a literate program, it is written as a work of literature, wherein code objects have equity in the narrative. When looking through the `pidgy` source code, we'll notice that `notebooks` are the substrate for the literate programs. The `notebook schema` defines specific containers for `markdown` and `"code"` cells; this duality motivates the use of `markdown` as the document formatting language.

What follows in an attempt at computable scientific literature, a programmatic paper.

1.1 The `notebook format` as a basis from literate programs.

The `notebook` flexible format for numerous input and output languages. The `notebook` relies on a consistent data structure, that may be validated be a `schema`, that serializes literate programs. The `notebook` has two primary cell types:

1. Markdown Cells

   Cells that are typically rendered as rich html, but have a plain-text representation.

2. Code Cells

   Cells that may be executed by a compiler external to a document.
   From the document perspective, code cells are forms that tangle to rich assemblages of mimetypes.

In `pidgy` programming, we look at notebooks as polyglot collages of input and output mimetypes.

```
return ("digraph {notebook->{documentation->{html pdf tex} python->{CLI extension␣
↪module tests}}}")
```

Short list of output formats

html

asciidoc

script

python

rst

notebook

markdown

custom

pdf

latex

slides

selectLanguage

1.2 Derived applications of pidgin programs.

```
@click.group()
def application()->None:
```

A successful `notebook` program could find uses outside of its interactive state as programs, documentation, or tests. `pidgy` programming includes a `click` command-line `application` to weave `notebooks` to other forms and tangle `notebooks` as source code.

```
def document(to, files):
```

The `document` command is an opinionated wrapper that converts notebooks to formatted python programs and readable documents.

```
exporter = nbconvert.get_exporter(to)
```

It uses the `nbconvert` library that transforms the `nbformat` into other projections.

```
if to in _CODE_FORMATS:
```

`pidgy` introduces a new opinion to the notebook where the input defines the output. In literate programming terms, we tangle the input and weave the output. The decoupling of the input & output means that proper python code maybe extracted from the `input`. `pidgy` includes `and isort` community conventions for formatting python to abide python styling guides.

```
    exporter = PythonExporter()
else:
```

With `pidgy`, we may consider a cell output to be the intended `display` set forth by an author. A string opinion `pidgy` documents is that the `input` is excluded from resulting document, where as typical approaches view all code as essential or not essential.

```
        exporter = exporter(exclude_input=True)

    for file in files: ...
```

(continues on next page)

(continued from previous page)

```
def run(files):
```

The `document` function demonstrates that `pidgy` may export `python` code. As a result the could be run as main scripts using the `runpy` modules.

```
    import pidgy, importnb, runpy
    with pidgy.reuse.pidgyLoader(), importnb.Notebook():
        for file in files: runpy.run_path(file)
        for module in modules: runpy.run_module(module)



@application.group()
def kernel():
```

`pidgy` is mainly designed to improve the interactive experience of creating literature in computational notebooks.

```
@kernel.command()
def install(user=False, replace=None, prefix=None):
    manager = jupyter_client.kernelspec.KernelSpecManager()
    path = str((pathlib.Path(__file__).parent / 'kernel' / 'spec').absolute())
    try:
        dest = manager.install_kernel_spec(path, 'pidgy')
    except:
        click.echo(F"System install was unsuccessful. Attempting to install the pidgy␣
↪kernel to the user.")
        dest = manager.install_kernel_spec(path, 'pidgy', True)
    click.echo(F"The pidgy kernel was install in {dest}")

@kernel.command()
def uninstall(user=True, replace=None, prefix=None):
    jupyter_client.kernelspec.KernelSpecManager().remove_kernel_spec('pidgy')
    click.echo(F"The pidgy kernel was removed.")
```

1.3 Deriving files from pidgin documents.

We can combine many syntaxes through markdown.

There are numerous tools that use the `notebook` format as an intermediate formats for different documents.

The original literate programming used latex as the sole export format where as the notebook recognizes quite a few formats:

nbconvert can generate 12 different formats from the files that abide the nbformat schema.

slides

markdown

rst

custom

python

script

notebook

asciidoc

html

pdf

selectLanguage

latex

```
class pidgyTranslate(nbconvert.preprocessors.Preprocessor):
```

Translate pidgy cells to pure python cells.

```
    def preprocess_cell(self, cell, resources, index, ):
        import pidgy
        tokenizer = pidgy.translate.Tokenizer()
        if cell['cell_type'] == 'code':
            cell['source'] = pidgy.imports.pidgy.transform_cell(''.join(cell['source
↪']))
        return cell, resources



class pidgyNormalize(nbconvert.preprocessors.Preprocessor):
```

Untangle a pidgy notebook into a normalized notebook that explicitly sepearting code and markdown cells. A normalized notebook can be imported by importnb.

```
def preprocess(self, nb, resources):
    new, tokens = nbformat.v4.new_notebook(), []
    for cell in nb.cells:
        for token in tokenizer.parse(''.join(cell.source)) if cell.cell_type == 'code
↪' else [{'type': 'paragraph', 'text': ''.join(cell.source)}]:
            new.cells.append((
                nbformat.v4.new_code_cell if token['type'] == 'code' else nbformat.v4.
↪new_markdown_cell
            )(token['text'].splitlines(True)))
    return nb, resources
```

1.4 Literature as the test

Intertextuallity emerges when the primary target of a program is literature. Some of the literary content may include `"code"` `objects` that can be tested to qualify the veracity of these dual signifiers.

`pidgy` documents are designed to be tested under multiple formal testing conditions. This is motivated by the `python`ic concept of documentation testing, or `doctest`ing, which in itself is a literate programming style. A `pidgy` document includes `doctest`, it verifies `notebook` `input`/`"output"`, and any formally defined tests are collected.

```
class pidgyModule(importnb.utils.pytest_importnb.NotebookModule):
```

pidgy provides a pytest plugin that works only on ".md.ipynb" files. The pidgy.kernel works directly with nbval, install the python packkage and use the –nbval flag. pidgy uses features from importnb to support standard tests discovery, and doctest discovery across all strings. Still working on coverage. The pidgyModule permits standard test discovery in notebooks. Functions beginning with "test_" indicate test functions.

```
    loader = pidgy.reuse.pidgyLoader



class pidgyTests(importnb.utils.pytest_importnb.NotebookTests):
```

if pidgy is install then importnb is.

```
modules = pidgyModule,
```

1.5 The pidgy shell-kernel model

The shell is the application either jupyterlab or jupyter notebook, the kernel determines the programming language. Below we design a just jupyter kernel that can be installed using

```
!pidgy kernel install
```

1.6 Building the pidgy extension

```
def load_ipython_extension(shell):
```

The pidgy implementation uses the IPython configuration and extension system to modify the interactive computing expierence in jupyter notebooks.

```
translate.load_ipython_extension(shell)
```

1. The primary function of pidgy is that it imports markdown as formal language for programming multiobjective literate programs. imports focuses on the indentification of "code" and not"code" that become python code.

   ```
   testing.load_ipython_extension(shell)
   ```

2. The pidgy specification promotes strong intertextuality between "code" and not"code" objects in a program. testing reinforces that efficacy of the "code" using documentation tests of doctest and "inline"+"code". pidgy uses the narrative a formal test for the program. These tests are executed interactively to ensure the veracity of "code" signs in the narrative.

   ```
   reuse.load_ipython_extension(shell)
   #debugging.load_ipython_extension(shell)
   outputs.load_ipython_extension(shell)
   ```

3. Literate computing in pidgy allows incremental development of "code" and the co-development of the documentation. pidgy interprets the input "code" as a display. pidgy uses a template language to transclude objects from code

CHAPTER 2

programming in `markdown and python`

```
def load_ipython_extension(shell):
    """
```

The `pidgy load_ipython_extension`'s primary function transforms the `jupyter notebooks` into a literate computing interfaces. `markdown` becomes the primary plain-text format for submitting code, and the `markdown` is translated to `python` source code before compilation. The implementation configures the appropriate features of the `IPython.InteractiveShell` to accomodate the interactive literate programming experience.

In this section, we'll implement a `shell.input_transformer_manager` that handles the logical translation of `markdown` to `python`. The translation maintains the source line numbers and normalizes the narrative relative to the source code. Consequently, introduces new syntaxes at the interfaces between `markdown and python`.

```
"""
pidgy_transformer = pidgyTransformer()
shell.input_transformer_manager = pidgy_transformer

"""
```

`IPython` provides configurable interactive `shell` properties. Some of the configurable properties control how `input` code is translated into valid source code. The `pidgy` translation is managed by a custom `IPython.core.inputtransformer2.TransformerManager`.

```
"""""""
>>> shell.input_transformer_manager
<...pidgyTransformer object...>

"""""""
```

The `shell.input_transformer_manager` applies string transformations to clean up the `input` to be valid `python`. There are three stages of line of transforms.

1. Cleanup transforms that operate on the entire cell `input`.

```
    """""""
    >>> shell.input_transformers_cleanup
    [<...leading_empty_lines...>, <...leading_indent...>, <...PromptStripper...>, ...]

    """""""
```

2. Line transforms that are applied the cell `input` with split lines. This is where `IPython` introduces their bespoke cell magic syntaxes.

```
    """""""
    >>> shell.input_transformer_manager.line_transforms
    [...<...cell_magic...>...]

    """""""
```

3. Token transformers that look for specific tokens at the like level. `IPython`'s default behavior introduces new symbols into the programming language.

```
    """""""
    >>> shell.input_transformer_manager.token_transformers
    [<...MagicAssign...SystemAssign...EscapedCommand...HelpEnd...>]

    """""""
```

After all of the `input` transformations are complete, the `input` should be valid source that `ast.parse, compile or shell.compile` may accept.

```
    """""""
    >>> shell.ast_transformers
    [...]

    """

    if not any(x for x in shell.ast_transformers if isinstance(x, ReturnYield)):
        shell.ast_transformers.append(ReturnYield())



class pidgyTransformer(IPython.core.inputtransformer2.TransformerManager):
    def pidgy_transform(self, cell: str) -> str:
        return self.tokenizer.untokenize(self.tokenizer.parse(''.join(cell)))

    def transform_cell(self, cell):
        return super().transform_cell(self.pidgy_transform(cell))

    def _rstrip_lines(self, lines):
        return '\n'.join(map(str.rstrip, ''.join(lines + ['']).splitlines())).
↪splitlines(True)

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.tokenizer = Tokenizer()
        self.line_transforms.append(demojize)
        self.line_transforms.append(self._rstrip_lines)

    def pidgy_magic(self, *text):
        """Expand the text to tokens to tokens and
```

(continues on next page)

(continued from previous page)

```
        compact as a formatted `"python"` code."""
        return IPython.display.Code(self.pidgy_transform(''.join(text)), language=
↪'python')



import ast
class ReturnYield(ast.NodeTransformer):
    def visit_FunctionDef(self, node): return node
    visit_AsyncFunctionDef = visit_FunctionDef
    def visit_Return(self, node):
        replace = ast.parse('''__import__('IPython').display.display()''').body[0]
        replace.value.args = node.value.elts if isinstance(node.value, ast.Tuple)␣
↪else [node.value]
        return ast.copy_location(replace, node)

    def visit_Expr(self, node):
        if isinstance(node.value, (ast.Yield, ast.YieldFrom)):  return ast.copy_
↪location(self.visit_Return(node.value), node)
        return node

    visit_Expression = visit_Expr



def demojize(lines, delimiters=('_', '_')):
    str = ''.join(lines)
    import tokenize, emoji, stringcase; tokens = []
    try:
        for token in list(tokenize.tokenize(
            __import__('io').BytesIO(str.encode()).readline)):
            if token.type == tokenize.ERRORTOKEN:
                string = emoji.demojize(token.string, delimiters=delimiters
                                        ).replace('-', '_').replace("'", "_")
                if tokens and tokens[-1].type == tokenize.NAME: tokens[-1] = tokenize.
↪TokenInfo(tokens[-1].type, tokens[-1].string + string, tokens[-1].start, tokens[-1].
↪end, tokens[-1].line)
                else: tokens.append(
                    tokenize.TokenInfo(
                        tokenize.NAME, string, token.start, token.end, token.line))
            else: tokens.append(token)
        return tokenize.untokenize(tokens).decode().splitlines(True)
    except BaseException: raise SyntaxError(str)



class Tokenizer(markdown.BlockLexer):
        """
```

Tokenize `input` text into `"code" and not "code"` tokens that will be translated into valid `python` source.

```
        """
        class grammar_class(markdown.BlockGrammar):
            doctest = doctest.DocTestParser._EXAMPLE_RE
            default_rules = "newline hrule block_code fences heading nptable lheading␣
↪block_quote list_block def_links def_footnotes table paragraph text".split()
```

(continues on next page)

(continued from previous page)

```
        def parse(self, text: str, default_rules=None) -> typing.List[dict]:
            if not self.depth: self.tokens = []
            with self: tokens = super().parse(whiten(text), default_rules)
            if not self.depth: tokens = self.normalize(text, tokens)
            return tokens

        def parse_doctest(self, m): self.tokens.append({'type': 'paragraph', 'text':␣
↪m.group(0)})

        def parse_fences(self, m):
            if m.group(2): self.tokens.append({'type': 'paragraph', 'text': m.
↪group(0)})
            else: super().parse_fences(m)

        def parse_hrule(self, m):
            self.tokens.append({'type': 'hrule', 'text': m.group(0)})

        def normalize(self, text, tokens):
            """Combine non-code tokens into contiguous blocks."""
            compacted = []
            while tokens:
                token = tokens.pop(0)
                if 'text' not in token: continue
                else:
                    if not token['text'].strip(): continue
                    block, body = token['text'].splitlines(), ""
                while block:
                    line = block.pop(0)
                    if line:
                        before, line, text = text.partition(line)
                        body += before + line
                if token['type']=='code':
                    compacted.append({'type': 'code', 'lang': None, 'text': body})
                else:
                    if compacted and compacted[-1]['type'] == 'paragraph':
                        compacted[-1]['text'] += body
                    else: compacted.append({'type': 'paragraph', 'text': body})
            if compacted and compacted[-1]['type'] == 'paragraph':
                compacted[-1]['text'] += text
            elif text.strip():
                compacted.append({'type': 'paragraph', 'text': text})
            return compacted

        depth = 0
        def __enter__(self): self.depth += 1
        def __exit__(self, *e): self.depth -= 1

        def untokenize(self, tokens: τ.List[dict], source: str = """""", last: int =0)␣
↪-> str:
            INDENT = indent = base_indent(tokens) or 4
            for i, token in enumerate(tokens):
                object = token['text']
                if token and token['type'] == 'code':
                    if object.lstrip().startswith(FENCE):

                        object = ''.join(''.join(object.partition(FENCE)[::2]).
↪rpartition(FENCE)[::2])
```

(continues on next page)

(continued from previous page)

```
                        indent = INDENT + num_first_indent(object)
                        object = textwrap.indent(object, INDENT*SPACE)

                    if object.lstrip().startswith(MAGIC):  ...
                    else: indent = num_last_indent(object)
                elif not object: ...
                else:
                    object = textwrap.indent(object, indent*SPACE)
                    for next in tokens[i+1:]:
                        if next['type'] == 'code':
                            next = num_first_indent(next['text'])
                            break
                    else: next = indent
                    Δ = max(next-indent, 0)

                    if not Δ and source.rstrip().rstrip(CONTINUATION).endswith(COLON):
                        Δ += 4

                    spaces = num_whitespace(object)
                    "what if the spaces are ling enough"
                    object = object[:spaces] + Δ*SPACE+ object[spaces:]
                    if not source.rstrip().rstrip(CONTINUATION).endswith(QUOTES):
                        object = quote(object)
                source += object

            for token in reversed(tokens):
                if token['text'].strip():
                    if token['type'] != 'code':
                        source = source.rstrip() + SEMI
                    break

            return source

for x in "default_rules footnote_rules list_rules".split():
    setattr(Tokenizer, x, list(getattr(Tokenizer, x)))
    getattr(Tokenizer, x).insert(getattr(Tokenizer, x).index('block_code'), 'doctest')

...
"""
```

```
"""
pidgy = pidgyTransformer()
```

A potential outcome of a `pidgy` program is reusable code.

Import pidgy notebooks as modules.

```
graphviz.Source(
```

digraph{rankdir=UD subgraph cluster_pidgy {label="new school" pidgy->{PYTHON MARKDOWN}} subgraph cluster_web {label="old school" WEB->{PASCAL TEX} }}

```
)
```

2.1 Weaving cells in pidgin programs

pidgin programming is an incremental approach to documents.

2.2 testing `"code"` in the `markdown` narrative.

```
import IPython as python, doctest, textwrap
pidgy= None
```

In literate programs, `"code"` is deeply entangled with the narrative. `"code"` object can signify meaning and can be validated through testing. `python` introduced the `doctest` literate programming convention that indicates some text in a narrative can be tested. `pidgy` extends the `doctest` opinion to the inline markdown code. Each time a `pidgy` cell is executed, the `doctests` and inline code are executed ensuring that any code in a `pidgy` program is valid.

```
def post_run_cell(result):
    result.runner = test_markdown_string(result.info.raw_cell, IPython.get_ipython(),␣
↪False, doctest.ELLIPSIS)

def load_ipython_extension(shell):
    unload_ipython_extension(shell)
    shell.events.register('post_run_cell', post_run_cell)



import doctest, contextlib, mistune as markdown, re, ast, __main__, IPython, operator
shell = IPython.get_ipython()
```

`test_markdown_string` extends the standard python `doctest` tools to inline code objects written in markdown.
This approach compliments are markdown forward programming language to test intertextual references between code and narrative.

```
INLINE = re.compile(
    markdown.InlineGrammar.code
    .pattern[1:]
    .replace('[\s\S]*', '?P<source>[\s\S]+')
    .replace('+)\s*', '{1,2})(?P<indent>\s{0})'),
)


(TICK,), SPACE = '`'.split(), ' '
```

(continues on next page)

(continued from previous page)

```
def test_markdown_string(str, shell=shell, verbose=False, compileflags=None):
    globs, filename = shell.user_ns, F"In[{shell.last_execution_result.execution_
↪count}]"
    runner = doctest.DocTestRunner(verbose=verbose, optionflags=compileflags)
    parsers = DocTestParser(runner), InlineDoctestParser(runner)
    parsers = {
        parser: doctest.DocTestFinder(verbose, parser).find(str, filename) for parser␣
↪in parsers
    }
    examples = sum([test.examples for x in parsers.values() for test in x], [])
    examples.sort(key=operator.attrgetter('lineno'))
    with ipython_compiler(shell):
        for example in examples:
            for parser, value in parsers.items():
                for value in value:
                    if example in value.examples:
                        with parser:
                            runner.run(doctest.DocTest(
                                [example], globs, value.name, filename, example.
↪lineno, value.docstring
                            ), compileflags=compileflags, clear_globs=False)
    shell.log.info(F"In[{shell.last_execution_result.execution_count}]: {runner.
↪summarize()}")
    return runner



@contextlib.contextmanager
def ipython_compiler(shell):
    def compiler(input, filename, symbol, *args, **kwargs):
        nonlocal shell
        return shell.compile(
            ast.Interactive(
                body=shell.transform_ast(
                    shell.compile.ast_parse(shell.transform_cell(textwrap.
↪indent(input, ' '*4)))
                ).body
            ),
            F"In[{shell.last_execution_result.execution_count}]",
            "single",
        )

    yield setattr(doctest, "compile", compiler)
    doctest.compile = compile
```

CHAPTER 3

`pidgy` metasyntax at language interfaces.

The combinations of document, programming, and templating languages provides unique syntaxes as the interfaces.

```
import mistune as markdown, IPython as python, pidgy
```

This is a code string

`pidgy` programming is a `markdown`-forward approach to programming, it extends computational to interactive literate programming environment. One feature `markdown` uses to identify `markdown.BlockGrammar.block_code` is indented code. `pidgy` starts here, all cells are `markdown` forward, and code is identified as indented code.

```
"This is a code string"
```

Some folks may prefer code fences and they may be used without a language specified.

```
"This is code"
```

```
"This is not code."
```

```
class DocStrings:
```

```
>>> assert DocStrings.__doc__.startswith('### Docstrings')
>>> DocStrings.function_docstring.__doc__
'`DocStrings.function_docstring`s appear as native docstrings, ...'


    def function_docstring():
```

`DocStrings.function_docstrings` appear as native docstrings, but render as `markdown`.

```
        ...




import doctest
```

```
>>> assert True
>>> print
<built-in function print>
>>> pidgy
<module...__init__.py'>
```

filters jinja docs