pidgy literate computing

Tony Fast

Apr 03, 2020

Contents

pidgy treats code as literature and programming as a literacy. It is an interactive programming workflow in Markdown that allows narrative and code to develop together.

pidgy has literary and computational qualities that:

- Publish documentation and render PDFs using the ReadTheDocs service.
- Make it installable from pip and conda.

```
pip install pidgy
```

- Formally test the literature and source code with Github Actions.
- Reusable on Binder.
- Import alternative source files into python like notebooks and markdown.

CHAPTER 1

The `pidgy` package and paper

`pidgy` is a fun way to program in [Markdown] in your favorite IDE (jupyter, nteract, colab, vscode) that can be reused as python modules, scripts, and applications.

Binder Documentation Status Python package PyPI - Python Version

```
pip install pidgy    # Install pidgy
```

`pidgy` has a few components to it:

- It is an interactive [Literate Computing] implementation of `IPython`
- A specification of a *potentially* polyglot approach for literate programming applied to other languages.
- A complete unit of computable scientific literate. It is written in a literate programming style with the literature as the primary outcome. *Read the `pidgy` paper*.

1.1 The pidgy shell and kernel

`pidgy` can be used as a native `jupyter` kernel in Jupyter, nteract, colab, and vscode. Install the kernel with

```
pidgy kernel install # install the pidgy kernel.
```

Or, in your standard Python shell, load the `pidgy` `IPython` extension.

1.2 Importing `pidgy` documents

`pidgy` uses the `importnb` machinery to import files into [Python] that are not native `".py"` files.

```
import pidgy
with pidgy.pidgyLoader(): ...
```

1.3 The `pidgy` CLI

```
Usage: pidgy [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  kernel
  render
  run       `pidgy` `run` makes it possible to execute `pidgy` documents as...
  template
  test      Formally test markdown documents, notebooks, and python files.
  to
```

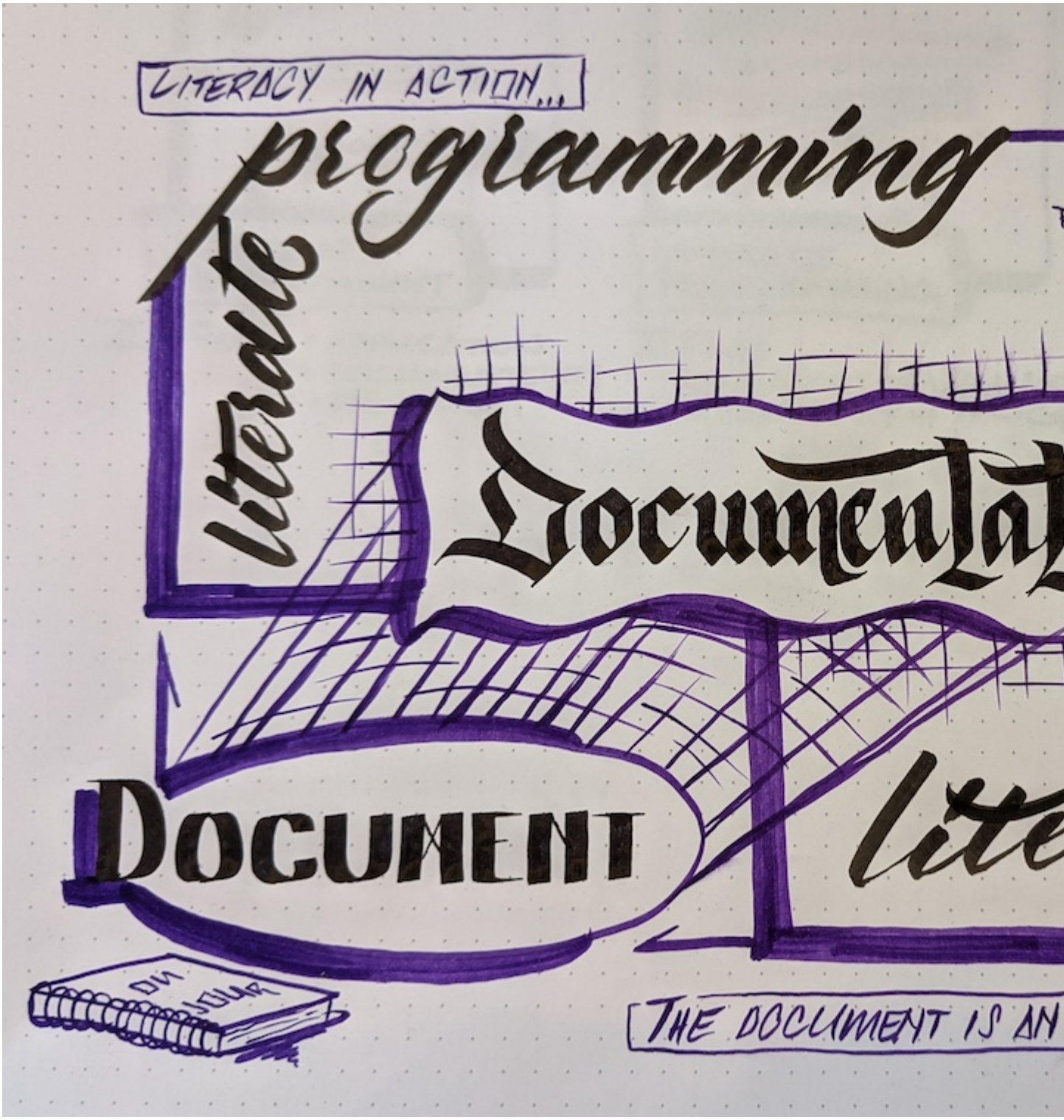
CHAPTER 2

pidgy paper

2.1 Introduction

A contemporary challenge for science is reprogramming literature for an information-rich scientific economy. Recently, science is troubled with an inability to reproduce scientific results from their published methods. This reproducibility crisis highlights the inability of natural language to bring meaning to data. In response, open science communities demonstrate that programming languages can supplement natural language as they are better equipped to communicate meaning in data. These progressive workflows disrupt traditional scientific publications with hypertext and hypermedia that are unbound from the constraints of physical media. These changes mean that modern scientific communications will necessitate different literacies when computability becomes a formal feature of scientific literature.

"Notebooks" have been adopted by differently-abled scientists and authors across disciplines to share stories supplemented with data and code. The availability of millions of open source notebooks demonstrates the potential of interleaved narrative, code, and hypermedia in computational thinking. The Jupyter Notebook ecosystem allows once-isolated scientific experiences to be shared in reproducible units through polyglot scientific languages. While the peer review process still requires a human reviewer to measure literary value, computational veracity can be automatically measured when an author adopts software engineering best practices.



Literate programming is a prescient, multi-lingual style of documentation that approaches programs as literary works. Minimally, a literate program is constrained to a document format and a programming language. Respectively, Donald Knuth's WEB implementation chose TeX and Pascal, while Jupyter notebooks use a pidgin of Markdown, TeX mathematical notation, tables, and syntax highlighting, and accept over 100 different programming languages. In a 2013 essay, Fernando Perez, author of IPython, describes literate computing as weaving interactive computation into scientific narratives, a different concern than the literary and computational qualities of the literate program. A strict

separation of these concerns creates inconsistencies, both for the author and reader, between the document and the program.

`pidgy` bridges literate programming and computing interactive programming experience that assigns Markdown as the primary programming language. The literate computing read-eval-print-loop allows composing documentation and software concurenntly. `pidgy` modifies the customizable `IPython` shell and kernel to:

1. before execution, tangle code line-for-line from Markdown to valid IPython syntax.
2. after execution, the input is woven into different representations:
    1. the outcome of formal `unittest and doctest`.
    2. template the Markdown using `jinja2` syntax and show a rich *Markdown* display.

Documents are composed interactively, and the introduction of literate program conventions for the REPL helps the ensure literary and computational integrity throughout the document. The `pidgy` is implemented as a literate program such that the `pidgy` paper and module are derived from the same sources, similarly, `pidgy` source code is used on the formal testing. `pidgy` demonstrates the flexability of literate code on different continuous integration systems for testing on Github Actions, publishing on Read the Docs, and packaging Pypi.

2.2 pidgy specification

2.2.1 The `pidgy` literate computing shell

https://ipython.readthedocs.io/en/stable/development/execution.html#execution-semantics

A powerful feature of the `jupyter` ecosystem is a generalized implementation of the Shell & Kernel model for interactive computing in interfaces like the terminal and notebooks. That is to say that different programming languages can use the same interfaces, `jupyter` supports over 100 languages now. The general ability to support different languages is possible because of configurable interfaces for the `IPython.InteractiveShell` and `ipykernel`.

```
import ipykernel.kernelapp, ipykernel.zmqshell, nbconvert, traitlets, pidgy, types,␣
↪pluggy, IPython, jinja2
class pidgyShell(ipykernel.zmqshell.ZMQInteractiveShell):
```

The `pidgy` shell is wrapper around the existing `IPython` shell experience. It explicitly defines the tangle and weave conventions of literate programming to each interactive computing execution. Once the shell is configured, it can be

reused as a `jupyter` kernel or `IPython` extension that supports the `pidgy` [Markdown]/[IPython] metalanguage and metasyntax.

```
    environment = traitlets.Any(nbconvert.exporters.TemplateExporter().environment)
```

`pidgy` specification

```
    @pidgy.specification(firstresult=True)
    def tangle(str:str)->str:
```

The `tangle` step operates on an input string that will become compiled source code. In a literate program, the source is written primarily in the documentation language and tangling converts it to the programming language. In `pidgy`, the tangle steps target valid `IPython` which is a superset of [Python], and requires further processing.

```
    input_transformers_post = traitlets.List([pidgy.tangle.demojize])
```

`pidgy` includes the ability the use emojis as valid python names through the existing `traitlets` configuration system.

```
    class pidgyManager(IPython.core.inputtransformer2.TransformerManager):
        def transform_cell(self, cell):
            shell = IPython.get_ipython()
            return super(type(self), self).transform_cell(
                (shell and hasattr(shell, 'manager') and shell.manager.hook.
↪tangle)(str=cell))

    input_transformer_manager = traitlets.Instance(pidgyManager, args=tuple())

    ast_transformers = traitlets.List([pidgy.tangle.ExtraSyntax(), pidgy.testing.
↪Definitions()])
```

Another feature of `IPython` is the ability to intercept [Abstract Syntax Tree]s and change their representation or capture metadata. After these transformations are applied, `IPython` compile the tree into a valid `types.CodeType`.

```
    @pidgy.specification
    def post_execute(self):
        ...


    @pidgy.specification
    def post_run_cell(self, result):
```

The weave step happens after execution, the tangle step happens before. Weaving only occurs if the input is computationally verified. It allows different representations of the input to be displayed. `pidgy` will implement templated Markdown displays of the input and formally test the contents of the input.

```
    def _post_run_cell(self, result):
        self.manager.hook.post_run_cell(result=result)

    def _post_exec(self):
        self.manager.hook.post_execute()


    enable_html_pager = traitlets.Bool(True)
    definitions = traitlets.List()
```

(continues on next page)

(continued from previous page)

```
    manager = traitlets.Instance('pluggy.PluginManager', args=('pidgy',))
    loaders = traitlets.Dict()
    weave = traitlets.Any()

    @traitlets.default('weave')
    def _default_weave(self): return pidgy.weave.Weave(self)
```

`pidgy` mixes the standard `IPython` configuration system and its own `pluggy` specification and implementation.

Initializing the `pidgy` shell

```
    def init_pidgy(self):
```

Initialize `pidgy` specific behaviors.

```
        self.manager.add_hookspecs(pidgyShell)
        for object in (
            pidgy.tangle, self.weave, pidgy.testing
        ):
```

The tangle and weave implementations are discussed in other parts of this document. Here we register each of them as `pluggy` hook implementations.

```
            self.manager.register(object)
        self.events.register("post_run_cell", types.MethodType(pidgyShell._post_run_
↪cell, self))
        self.events.register("post_execute", types.MethodType(pidgyShell._post_exec,␣
↪self))


        if pidgy.pidgyLoader not in self.loaders:
```

`pidgy` enters a loader context allowing [Markdown] and notebook files to be used permissively as input.

```
            self.loaders[pidgy.pidgyLoader] = pidgy.pidgyLoader().__enter__()
```

It also adds a few extra features to the shell.

```
        self.user_ns["shell"] = self
        self.user_ns.update({k: v for k, v in vars(IPython.display).items()
            if pidgy.util.istype(v, IPython.core.display.DisplayObject)
        })
```

and allows json syntax as valid python input.

```
        pidgy.tangle.init_json()
        pidgy.magic.load_ipython_extension(self)

    def __init__(self, *args, **kwargs):
```

Override the initialization of the conventional IPython kernel to include the pidgy opinions.

```
        super().__init__(*args, **kwargs)
        self.init_pidgy()
```

`pidgy` extension.

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

The pidgy kernel makes it easy to access the pidgy shell, but it can also be used an IPython extension.

```
        shell.add_traits(manager=pidgyShell.manager, loaders=pidgyShell.loaders,␣
↪definitions=pidgyShell.definitions, weave=pidgyShell.weave)
        shell._post_run_cell = types.MethodType(pidgyShell._post_run_cell, shell)
        shell._post_exec = types.MethodType(pidgyShell._post_exec, shell)
        pidgyShell.init_pidgy(shell)
        shell.input_transformer_manager = pidgyShell.input_transformer_manager.
↪default_value
```

```
    def unload_ipython_extension(self):
        self.events.unregister("post_run_cell", self._post_run_cell)
        self.events.unregister("post_run_cell", pidgy.weave.post_run_cell)
        loader = self.loaders.pop(pidgy.pidgyLoader)
        if loader is not None:
            loader.__exit__(None, None, None)


load_ipython_extension = pidgyShell.load_ipython_extension
unload_ipython_extension = pidgyShell.unload_ipython_extension
```

2.3 pidgy implementation

2.3.1 Translating Markdown to Python

A primary translation is literate programming is the tangle step that converts the literate program into the programming language. The 1979 implementation converts `".WEB"` files to valid pascal - `".PAS"` - files. The `pidgy` approach begins with *Markdown* files and proper *Python* files as the outcome. The rest of this document configures how [IPython] acknowledges the transformation and the heuristics the translate *Markdown* to *Python*.

[1]:
```
    import typing, mistune, IPython, pidgy.util
    __all__ = 'tangle', 'Tangle'
```

The `pidgy` tangle workflow has three steps:

1. Block-level lexical analysis to tokenize *Markdown*.
2. Normalize the tokens to compacted `"code" and not "code"` tokens.
3. Translate the normalized tokens to a string of valid *Python* code.

[2]:
```
    @pidgy.implementation
    def tangle(str:str)->str:
        translate = Tangle()
        return translate.stringify(translate.parse(''.join(str)))
```

[3]:
```
    class pidgyManager(IPython.core.inputtransformer2.TransformerManager):
        def transform_cell(self, cell): return super(type(self), self).transform_
↪cell(tangle(str=cell))
```

Block level lexical analysis.

`pidgy` uses a modified `mistune.BlockLexer` to create block level tokens for a [Markdown] source. A specific `pidgy` addition is the addition off a `doctest` block object, `doctest` are testable strings that are ignored by the tangle step. The tokens are to be normalized and translated to [Python] strings.

BlockLexer

[4]:
```
    class BlockLexer(mistune.BlockLexer, pidgy.util.ContextDepth):
        class grammar_class(mistune.BlockGrammar):
            doctest = __import__('doctest').DocTestParser._EXAMPLE_RE
            block_code = __import__('re').compile(r'^((?!\s+>>>\s) {4}[^\n]+\n*)+')
            default_rules = "newline hrule block_code fences heading nptable lheading␣
↪block_quote list_block def_links def_footnotes table paragraph text".split()

        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(dict(type='hrule', text=m.
↪group(0)))

        def parse_def_links(self, m):
            super().parse_def_links(m)
            self.tokens.append(dict(type='def_link', text=m.group(0)))

        def parse_front_matter(self): ...
        def parse(self, text: str, default_rules=None, normalize=True) -> typing.
↪List[dict]:
            front_matter = None
            if not self.depth:
                self.tokens = []
                if text.strip() and text.startswith('---\n') and '\n---\n' in text[4:
↪]:
                    front_matter, sep, text = text[4:].partition('---\n')
                    front_matter = {'type': 'front_matter', 'text': F"\n{front_matter}
↪"}

            with self: tokens = super().parse(pidgy.util.whiten(text), default_rules)
            if normalize and not self.depth: tokens = normalizer(text, tokens)
            if front_matter: tokens.insert(0, front_matter)
            return tokens
```

Normalizing the tokens

Tokenizing [Markdown] typically extracts conventions at both the block and inline level. Fortunately, `pidgy`'s translation is restricted to block level [Markdown] tokens, and mitigating some potential complexities from having opinions about inline code while tangling.

normalizer

[5]:
```
    def normalizer(text, tokens):
        compacted = []
```

(continues on next page)

(continued from previous page)

```
        while tokens:
            token = tokens.pop(0)
            if 'text' not in token: continue
            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})
            elif 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})
        # Deal with front matter
        if compacted and compacted[0]['text'].startswith('---\n') and '\n---' in␣
↪compacted[0]['text'][4:]:
            token = compacted.pop(0)
            front_matter, sep, paragraph = token['text'][4:].partition('---')
            compacted = [{'type': 'front_matter', 'text': F"\n{front_matter}"},
                         {'type': 'paragraph', 'text': paragraph}] + compacted
        return compacted
```

Flattening the tokens to a [Python] string.

The tokenizer controls the translation of markdown strings to python strings. Our major constraint is that the Markdown input should retain line numbers.

Flatten

[6]:
```
    class Tangle(BlockLexer):
        def stringify(self, tokens: typing.List[dict], source: str = """""", last:␣
↪int =0) -> str:
            import textwrap
            INDENT = indent = pidgy.util.base_indent(tokens) or 4
            for i, token in enumerate(tokens):
                object = token['text']
                if token and token['type'] == 'code':
                    if object.lstrip().startswith(pidgy.util.FENCE):

                        object = ''.join(''.join(object.partition(pidgy.util.FENCE)[::
↪2]).rpartition(pidgy.util.FENCE)[::2])
                        indent = INDENT + pidgy.util.num_first_indent(object)
                        object = textwrap.indent(object, INDENT*pidgy.util.SPACE)

                    if object.lstrip().startswith(pidgy.util.MAGIC):  ...
                    else: indent = pidgy.util.num_last_indent(object)
                elif token and token['type'] == 'front_matter':
                    object = textwrap.indent(
```

(continues on next page)

(continued from previous page)

```
                        F"locals().update(__import__('ruamel.yaml').yaml.safe_load(
↪{pidgy.util.quote(object)}))\n", indent*pidgy.util.SPACE)

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

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

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

            # add a semicolon to the source if the last block is code.
            for token in reversed(tokens):
                if token['text'].strip():
                    if token['type'] != 'code':
                        source = source.rstrip() + pidgy.util.SEMI
                    break

            return source
```

Append the lexer for nested rules.

[7]:
```
    for x in "default_rules footnote_rules list_rules".split():
        setattr(BlockLexer, x, list(getattr(BlockLexer, x)))
        getattr(BlockLexer, x).insert(getattr(BlockLexer, x).index('block_code'),
↪'doctest')
        if 'block_html' in getattr(BlockLexer, x):
            getattr(BlockLexer, x).pop(getattr(BlockLexer, x).index('block_html'))
    del x
```

More `pidgy` langauge features

`pidgy` experiments extra language features for python, using the same system that IPython uses to add features like line and cell magics.

[1]:
```
    import ast, pidgy, IPython
```

Recently, IPython introduced a convention that allows top level await statements outside of functions. Building of this convenience, `pidgy` allows for top-level **return** and **yield** statements. These statements are replaced with the an IPython display statement.

[2]:
```
    class ExtraSyntax(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
```

We know naming is hard, there is no point focusing on it. `pidgy` allows authors to use emojis as variables in python. They add extra color and expression to the narrative.

[3]:
```
    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()
        except BaseException: raise SyntaxError(str)
```

[ ]:
```
    def init_json():
        import builtins
        builtins.yes = builtins.true = True
        builtins.no = builtins.false = False
        builtins.null = None
```

2.3.2 Import Markdown files and notebooks

Literate `pidgy` programs are reusable as [Python] scripts and modules. These features are configured by inheriting features from `importnb` that customize the [Python] import system to discover/load alternative source files. `pidgy` treats [Python], [Markdown], and [Notebook] files as python source.

`sys.meta_path and sys.path_hooks`

[1]:
```
    __all__ = 'pidgyLoader',
    import pidgy, IPython, importnb
```

`get_data` determines how a file is decoding from disk. We use it to make an escape hatch for markdown files otherwise we are importing a notebook.

[2]:
```
    def get_data(self, path):
        if self.path.endswith('.md'): return self.code(self.decode())
        return super(pidgyLoader, self).get_data(path)
```

The `code` method tangles the [Markdown] to [Python] before compiling to an [Abstract Syntax Tree].

[3]:
```
    def code(self, str):
        for callable in (self.transformer_manager.transform_cell,
                         pidgy.tangle.demojize):
            str = ''.join(callable(''.join(str)))
        return str
```

The `visit` method allows custom [Abstract Syntax Tree] transformations to be applied.

[4]:
```
    def visit(self, node):
        return pidgy.tangle.ExtraSyntax().visit(node)
```

Attach these methods to the `pidgy` loader.

Only [Python] files and common flavored notebooks may be used as source code before the `pidgyLoader` is defined. Once the `pidgyLoader` is defined [Markdown] becomes a new source target for [Python] and [Notebook]s bearing the `".md.ipynb"` extension are consumed specially as `pidgy` flavored documents.

[5]:
```
    class pidgyLoader(importnb.Notebook):
        extensions = ".py.md .md .md.ipynb".split()
        transformer_manager = pidgy.tangle.pidgyManager()
        code = code
        visit = visit
        get_source = get_data = get_data
```

[ ]:
```

```

2.3.3 Render and template output source

In literate programming, the input is representative of a published form. The original target for the WEB programming implementation is the Device Independent Format used by Latex, and with the ability to target PDF. [Markdown] is the `pidgy` document language. It is a plain text formatting syntax that has canonical representations in HTML.

An important feature of interactive computing in the browser is access to rich display object provided by HTML and Javascript. `pidgy` takes advantage of the ability to include hypermedia forms that enhance and support computational narratives.

```
import dataclasses, IPython, pidgy

@dataclasses.dataclass(unsafe_hash=True)
class Weave:

    exporter = __import__('nbconvert').exporters.TemplateExporter()
```

The `Weave` class controls the display of `pidgy` outputs.

```
    shell: object

```

(continues on next page)

(continued from previous page)

```
    @pidgy.implementation
    def post_run_cell(self, result):
```

Show the woven output.

```
        text = pidgy.util.strip_front_matter(result.info.raw_cell)
        lines = text.splitlines() or ['']
        if not lines[0].strip(): return pidgy.util.html_comment(text)
        IPython.display.display(IPython.display.Markdown(self.render(text)))


    def render(self, text):
        import builtins, operator
        try:
```

Try to replace any jinja templates with information in the current namespace and show the rendered view.

```
            template = self.exporter.environment.from_string(text, globals={
                **vars(builtins), **vars(operator),
                **getattr(self.shell, 'user_ns', {})})
            text = template.render()
        except BaseException as Exception:
            IPython.get_ipython().showtraceback((type(Exception), Exception,␣
↪Exception.__traceback__))

        return text
```

2.3.4 Interactive formal testing

Testing is something we added because of the application of notebooks as test units.

A primary use case of notebooks is to test ideas. Typically this in informally using manual validation to qualify the efficacy of narrative and code. To ensure testable literate documents we formally test code incrementally during interactive computing.

```
def make_test_suite(*objects: typing.Union[
    unittest.TestCase, types.FunctionType, str
], vars, name) -> unittest.TestSuite:
```

The interactive testing suite execute `doctest and unittest` conventions for a flexible interface to verifying the computational qualities of literate programs.

```
    suite, doctest_suite = unittest.TestSuite(), doctest.DocTestSuite()

    for object in objects:
        if isinstance(object, type) and issubclass(object, unittest.TestCase):
            suite.addTest(unittest.defaultTestLoader.loadTestsFromTestCase(object))
        elif isinstance(object, str):
            doctest_suite.addTest(doctest.DocTestCase(
            doctest.DocTestParser().get_doctest(object, vars, name, name, 1), doctest.
↪ELLIPSIS))
            doctest_suite.addTest(doctest.DocTestCase(
            InlineDoctestParser().get_doctest(object, vars, name, name, 1),␣
↪checker=NullOutputCheck))
        elif inspect.isfunction(object):
```

(continues on next page)

(continued from previous page)

```
            suite.addTest(unittest.FunctionTestCase(object))

    doctest_suite._tests and suite.addTest(doctest_suite)
    return suite

@pidgy.implementation
def post_run_cell(result):
        shell = IPython.get_ipython()
        globs, filename = shell.user_ns, F"In[{shell.last_execution_result.execution_
↪count}]"

        if not (result.error_before_exec or result.error_in_exec):
            definitions = []
            with ipython_compiler(shell):
                while shell.definitions:
                    definition = shell.definitions.pop(0)
                    object = shell.user_ns.get(definition, None)
                    if definition.startswith('test_') or pidgy.util.istype(object,␣
↪unittest.TestCase):
                        definitions.append(object)
                result = run(make_test_suite(result.info.raw_cell, *definitions,␣
↪vars=shell.user_ns, name=filename), result)

class Definitions(ast.NodeTransformer):
    def visit_FunctionDef(self, node):
        shell = IPython.get_ipython()
        shell and shell.definitions.append(node.name)
        return node
    visit_ClassDef = visit_FunctionDef

def run(suite: unittest.TestCase, cell) -> unittest.TestResult:
        result = unittest.TestResult(); suite.run(result)
        if result.failures:
            msg = '\n'.join(msg for text, msg in result.failures)
            msg = re.sub(re.compile("<ipython-input-[0-9]+-\S+>"), F'In[{cell.
↪execution_count}]', clean_doctest_traceback(msg))
            sys.stderr.writelines((str(result) + '\n' + msg).splitlines(True))
            return result

@contextlib.contextmanager
def ipython_compiler(shell):
```

We'll have to replace how `doctest` compiles code with the `IPython` machinery.

```
    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)
```

(continues on next page)

(continued from previous page)

```
    doctest.compile = compile

def clean_doctest_traceback(str, *lines):
    str = re.sub(re.compile("""\n\s+File [\s\S]+, line [0-9]+, in␣
↪runTest\s+raise[\s\S]+\([\s\S]+\)\n?"""), '\n', str)
    return re.sub(re.compile("Traceback \(most recent call last\):\n"), '', str)
```

```
class NullOutputCheck(doctest.OutputChecker):
    def check_output(self, *e): return True

class InlineDoctestParser(doctest.DocTestParser):
    _EXAMPLE_RE = re.compile(r'`(?P<indent>\s{0})'
r'(?P<source>[^`].*?)'
r'`')
    def _parse_example(self, m, name, lineno): return m.group('source'), None, "...",␣
↪None
```

2.3.5 `pidgy` metasyntax

[1]:
```
    import pidgy, IPython, jinja2, doctest

`pidgy` not only allows the [Markdown] and [Python] to cooperate in a document,
metasyntaxes emerge at the interface between the language.
```

```
import pidgy, IPython, jinja2, doctest
```

pidgy not only allows the [Markdown] and [Python] to cooperate in a document, metasyntaxes emerge at the interface between the language.

Markdown is the primary language

[2]:
```
`pidgy` considers [Markdown] indented code blocks and language free code fences
as valid [Python] code while every other object is represented as a triple
quoted block string.

    print("Indented blocks are always code like in literate coffeescript.")
```

```
Indented blocks are always code like in literate coffeescript.
```

pidgy considers [Markdown] indented code blocks and language free code fences as valid [Python] code while every other object is represented as a triple quoted block string.

```
print("Indented blocks are always code like in literate coffeescript.")
```

Executing code.

[3]:
```
There are two specific to ensure that code is executed in `pidgy`.

### Indented code.
```

(continues on next page)

(continued from previous page)

```
Like in the prior cell, an indented code block is a specific token in Markdown
that `pidgy` recognizes canonically as code.

    "This is code" # because of the indent.

### Code fences.

```
"I am code because no language is specified."
```

Ignoring code.

Include a language with the code fence to skip code execution.

```alangage
Add alanguage specification to code fence to ignore its input.
```

Or, use html tags.

<pre><code>
I am explicit HMTL.
</code></pre>
```

There are two specific to ensure that code is executed in `pidgy`.

Indented code.

Like in the prior cell, an indented code block is a specific token in Markdown that `pidgy` recognizes canonically as code.

```
"This is code" # because of the indent.
```

Code fences.

```
"I am code because no language is specified."
```

Ignoring code.

Include a language with the code fence to skip code execution.

```
Add alanguage specification to code fence to ignore its input.
```

Or, use html tags.

Testing code

[4]:
```
`pidgy` recognizes doctests, a literate programming approach to testing, in the input␣
↪and executes them in a formal unittest
testing suite. `doctest` are identified by the `">>>"` line prefix.

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

`pidgy` recognizes doctests, a literate programming approach to testing, in the input and executes them in a formal unittest testing suite. `doctest` are identified by the `">>>"` line prefix.

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

Weaving and templating code

[5]:
```
`pidgy` permits the popular `jinja2` templating syntax.  Any use of
templates references <code>{% raw %}{{}}{% endraw %}</code> will be filled in with
information from the current namespace.

There is a variable `foo` with the value <code>{{foo}}</code>.

    foo = 20
```

`pidgy` permits the popular `jinja2` templating syntax. Any use of templates references {{}} will be filled in with information from the current namespace.

There is a variable `foo` with the value 20.

```
foo = 20
```

Suppressing the weave output.

[6]:
```


`pidgy` will not render any input beginning with a blank line.
```

Emergent language features

[7]:
```
Interleaving Markdown and Python results in natural metasyntaxes that allow
`pidgy` authors to write programs that look like documentation.

### Docstrings.

[Markdown] that follows function and class definitions are wrapped as block strings
```

(continues on next page)

(continued from previous page)

```
and indented according `pidgy`'s heuristics. What results is the [Markdown]
represents the docstring.

    def my_function():

`my_function` demonstrates how docstrings are defined.

    class MyClass:

The same goes for class definitions.

    ...
    >>> my_function.__doc__
    '`my_function` demonstrates how ...'
    >>> MyClass.__doc__
    'The same goes for class definitions.'
```

Interleaving Markdown and Python results in natural metasyntaxes that allow `pidgy` authors to write programs that look like documentation.

Docstrings.

[Markdown] that follows function and class definitions are wrapped as block strings and indented according `pidgy`'s heuristics. What results is the [Markdown] represents the docstring.

```
def my_function():
```

`my_function` demonstrates how docstrings are defined.

```
class MyClass:
```

The same goes for class definitions.

```
...
>>> my_function.__doc__
'`my_function` demonstrates how ...'
>>> MyClass.__doc__
'The same goes for class definitions.'
```

```
# NBVAL_SKIP
```

Interactive Testing

Failures are treated as natural outputs of the documents. Tests may fail, but parts of the unit may be reusable.

```
def test_functions_start_with_test():
    assert False, "False is not True"
    assert False is not True
...
```

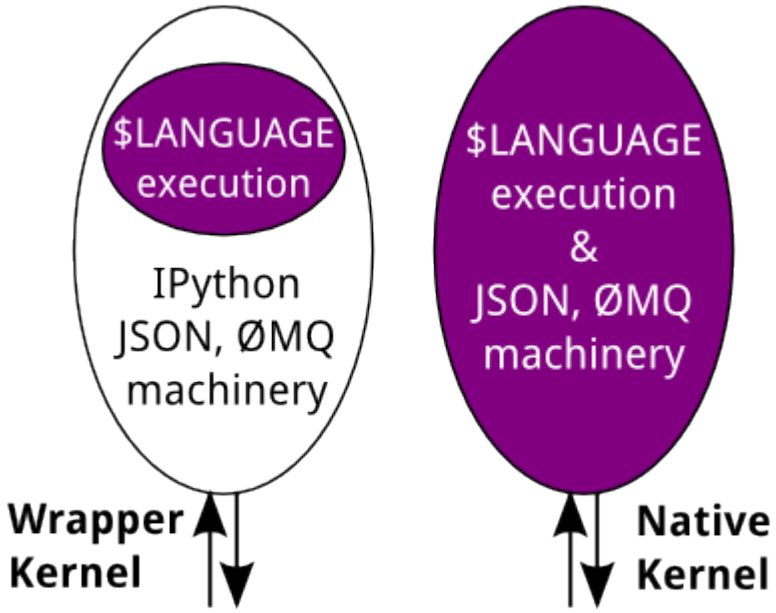
2.4 pidgy applications

2.4.1 `pidgy` kernel

A kernel provides programming language support in Jupyter. IPython is the default kernel. Additional kernels include R, Julia, and many more.

- `jupyter` kernel definition

`pidgy` is a wrapper kernel around the existing `ipykernel and IPython.InteractiveShell`.




```
import IPython, ipykernel.ipkernel, ipykernel.kernelapp, pidgy, traitlets, ipykernel.
↪kernelspec, ipykernel.zmqshell, pathlib

class pidgyKernel(ipykernel.ipkernel.IPythonKernel):
```

The `pidgy` kernel specifies to `jupyter` how it can be used as a native kernel from the launcher or notebook. It specifies which shell class to use.

```
    shell_class = traitlets.Type('pidgy.shell.pidgyShell')
    loaders = traitlets.Dict()
    _last_parent = traitlets.Dict()
    current_cell_id = traitlets.Unicode()
    current_cell_ids = traitlets.Set()

    def init_metadata(self, *args, **kwargs):
```

The is some important data captured in the initial we'll expose for later.

```
        return super().init_metadata(*args, **kwargs)

    def do_inspect(self, code, cursor_pos, detail_level=0):
```

The kernel is where the inspection can be customized. `pidgy` adds the ability to use the inspector as Markdown rendering tool.

```
        if code[cursor_pos-3:cursor_pos] == '!!!':
            if code[cursor_pos-6:cursor_pos] == '!!!'*2:
                self.shell.run_cell(code, silent=True)
            return self.markdown_result(self.shell.weave.format_output(code))
        result = super().do_inspect(code, cursor_pos, detail_level)
        if not result['found']: return self.markdown_result(code)
        return result

    def markdown_result(self, code):
        return dict(found=True, status='ok', metadata={}, data={'text/markdown': code}
↪)

    def do_complete(self, code, cursor_pos):
```

The kernel even allows the completion system to be modified.

```
        return super().do_complete(code, cursor_pos)
```

`pidgy` kernel installation

```
def install():
```

`install` the pidgy kernel.

```
    import jupyter_client, click
    manager = jupyter_client.kernelspec.KernelSpecManager()
    path = str((pathlib.Path(__file__).parent / 'kernelspec').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}")
```

```
def uninstall():
```

`uninstall` the kernel.

```
    import jupyter_client, click
    jupyter_client.kernelspec.KernelSpecManager().remove_kernel_spec('pidgy')
    click.echo(F"The pidgy kernel was removed.")
```

```
def start(f:str=""):
```

Launch a `pidgy` kernel applications.

```
    ipykernel.kernelapp.IPKernelApp.launch_instance(connection_file=f, kernel_
↪class=pidgyKernel)
...
```

2.4.2 Export documents to other formats.

Literate programs can be translated into many formats. Commonly markup langauges focus on translation to other markup languages, we add an extra ability to convert markup to source code.

```
import pidgy, pathlib, typing, textwrap, IPython
try: from . import util
except: import util
```

We can reuse existing nbconvert machinery if we expand every file to a notebook.

```
def file_to_nb(file: pathlib.Path) -> "nbformat.NotebookNode":
    import nbformat
    if file.suffix in {'.md', '.markdown'}:
        return nbformat.v4.new_notebook(cells=[nbformat.v4.new_code_cell(file.read_
↪text())])
    return nbformat.reads(file.read_text(), 4)
```

A notebook can also be flattened.

```
def flattennb(nb: typing.Union[str, "nbformat.NotebookNode"]):
    if isinstance(nb, str): return nb
    return [textwrap.indent(''.join(x.source), x.cell_type != 'code' and '# ' or '')␣
↪for x in nb.cells]

def to_markup(input: typing.Union[str, "nbformat.NotebookNode"], exporter: "nbconvert.
↪Exporter") -> str:
    return exporter.from_notebook_node(input)[0]

def to_python(input, tangle = pidgy.tangle.pidgyManager()):
    import black, isort
    code = pidgy.loader.tangle(flattennb(input))
    code = isort.SortImports(file_contents=code).output
    code = black.format_str(code, mode=black.FileMode(line_length=100))
    return code

def convert(*files, to: {'python', 'markdown'}, write: bool=False):
    import nbconvert
    exporter = nbconvert.get_exporter(to)()
    for file in util.yield_files(files):
        nb = file_to_nb(file)
        if to =='python':
            body = '\n'.join(to_python(''.join(getattr(x, 'source', []))) for x in nb.
↪cells)
        else:
            body = to_markup(nb, exporter)
        if write:
            new = pathlib.Path(file).with_suffix(dict(python='.py', markdown='.md
↪')[to])
            new.write_text(body)
            __import__('click').echo(F"{new} created.")
        else:
            __import__('click').echo(body)
```

https://stackoverflow.com/questions/34643620/how-can-i-split-my-click-commands-each-with-a-set-of-sub-commands-into-multipl

2.4.3 Scripting with literate programs.

Since `pidgy` is based on [Python], derived pidgy documents can be used as scripts.

A `pidgy` program executed as the **main** program has similar state to the running notebook and it introduces the file object.

`pidgy` is based on [Python], a scripting language, therefore it should be possible execute markdown as scripts.

```
import types, pidgy, ast, runpy, importlib
__all__ = 'run', 'render', 'Runner'


def run(object: str, **globals) -> dict:
```

`run` executes a literate document as a script.

```
    return Runner(object).run(**globals)

def render(object: str, **globals) -> dict:
```

`render` executes a templated document.

```
    return Runner(object).render(**globals)

...

class Runner(pidgy.pidgyLoader):
```

A script `Runner` for `pidgy` documents based off the `importnb` machinery.

```
    def __init__(self, name, path=None, *args, **kwargs):
        if path is None: path = name
        super().__init__(name, path, *args, **kwargs)
    def visit(self, node):
        node = super().visit(node)
        body, annotations = ast.Module([]), ast.Module([])
        while node.body:
            element = node.body.pop(0)
            if isinstance(element, ast.AnnAssign) and element.target.id[0].islower():
                try:
                    if element.value:
                        ast.literal_eval(element.value)
                    annotations.body.append(element)
                    continue
                except: ...
            if isinstance(element, (ast.Import, ast.ImportFrom)):
                annotations.body.append(element)
            body.body.append(element)
        self.arg_code = compile(annotations, self.path, 'exec')
        return body

    def create_module(loader, spec=None):
```

When the module is created. Compile the source to code to discover arguments in the code.

```
        if spec is None:
            spec = importlib.util.spec_from_loader(loader.name, loader)
```

(continues on next page)

(continued from previous page)

```
            module = super().create_module(spec)
        loader.main_code = loader.get_code(loader.name)
        runpy._run_code(loader.arg_code, vars(module), {}, '__main__', spec, None,␣
↪None)
        return module

    def exec_module(loader, module=None, **globals):
        module = module or loader.create_module()
        vars(module).update(globals)
        runpy._run_code(loader.main_code, vars(module), {}, '__main__', module.__spec_
↪_, None, None)
        return module

    def run(loader, **globals):
        return loader.exec_module(**globals)

    def render(loader, **globals):
        return loader.format(loader.run(**globals))

    def cli(loader):
        import pidgy.autocli, click
        module = loader.create_module()
        def main(verbose: bool=True, **globals):
            nonlocal module
            try:
                loader.exec_module(module, **globals)
                verbose and click.echo(pidgy.util.ansify(loader.format(module)))
            except SystemExit: ...

        pidgy.autocli.command_from_decorators(main,
                                              click.option('--verbose/--silent',␣
↪default=True),
                                              *pidgy.autocli.decorators_from_
↪module(module)).main()

    def format(loader, module):
        import nbconvert, operator, builtins
        if loader.path.endswith(('.py', '.md', '.markdown')):
            return nbconvert.TemplateExporter().environment.from_string(
                pidgy.util.strip_front_matter(
                    pidgy.util.strip_html_comment(
                        pidgy.util.strip_shebang(
                            loader.decode()))))
            ).render({
                **vars(operator), **vars(builtins),
                **vars(module)}).rstrip() + '\n'

...
```

shebang statements in literate programs.

A feature of `pidgy` markdown files, not notebook files, is that a shebang statement can be included at the beginning to indicate how a document is executed.

Some useful shebang lines to being pidgy documents with.

```
#!/usr/bin/env pidgy run
#!/usr/bin/env python -m pidgy run
#!/usr/bin/env python -m pidgy render
#!/usr/bin/env pidgy render
```

2.4.4 Formally testing literate programs

```
import pidgy, pytest, doctest, importnb.utils.pytest_importnb
```

Literate documents can be motivated by the need to test a concept. In a fact, a common use case of notebooks is that they interactively test units of thought. Often the thought of reusability is an after thought.

`pidgy` documents are meant to be treated as test objects. In fact, the `pidgy` test suite executed by `pytest` through Github Actions uses `pidgy` notebooks (ie. documents with the `".md" or ".md.ipynb"` extension). `pidgy` supplies its own `pytest` extensions, and it uses `nbval` and the `pytest`"–doctest-modules"`flag. With these conditions we discover pytest conventions, unitests, doctests, and options cell input output validated. Ultimately,`pidgy‘ documents may represent units of literate that double as formal test objects.

The document accessed by the `"pytest11"` console_script and includes the extension with a pytest runner.

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

The `pidgyModule` derives from an existing `pytest` extension that extracts formal tests from `notebooks` as if they were regular python files. We'll use the `pidgy.pidgyLoader` to load Markdown-forward documents as python objects.

```
    loader = pidgy.pidgyLoader

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

`pidgyTests` makes sure to include the alternative source formats to tangle to python executions.

```
    modules = pidgyModule,
```

2.4.5 `pidgy` command line interface

> **Eat Me, Drink Me, Read Me.**

```
import IPython, pidgy, pathlib, typing, click, functools, contextlib, sys, types

with pidgy.pidgyLoader():
    try: from . import kernel, autocli, runpidgy, util, export, weave
    except: import kernel, autocli, runpidgy, util, export, weave
```

```
def run(ctx, ref: str):
```

`pidgy run` executes `pidgy` documents as programs.

```
    import pidgy, click
    click.echo(F"Running {ref}.")
    with pidgy.util.sys_path(), pidgy.util.argv(*([ref] + ctx.args)):
        runpidgy.run(ref)
```

(continues on next page)

(continued from previous page)

```
def template(ctx, ref: str, no_show:bool=False):
```

`pidgy template` executes `pidgy` documents as programs and publishes the templated results.

```
    import pidgy, click
    with pidgy.util.sys_path(), pidgy.util.argv(*([ref] + ctx.args)):
        data = pidgy.runpidgy.render(ref)
        if not no_show: click.echo(pidgy.util.ansify(data))

def to(to:{'markdown', 'python'}, files: typing.List[pathlib.Path], write:bool=False):
```

Convert pidgy documents to other formats.

```
    pidgy.export.convert(*files, to=to, write=write)
```

```
def test(ctx, files: list):
```

Formally test markdown documents, notebooks, and python files.

```
     import pytest
     pytest.main(ctx.args+['--doctest-modules', '--disable-pytest-warnings
↪']+list(files))
```

```
application = autocli.autoclick(
    run, test, to, template,
    autocli.autoclick(
        pidgy.kernel.install, pidgy.kernel.uninstall, pidgy.kernel.start, group=click.
↪Group("kernel")
    ),
    context_settings=dict(allow_extra_args=True, ignore_unknown_options=True),
)
```

2.4.6 Discussion

Markdown for literate programming

Alternative source files

[1]:
```
    import pidgy, pandas, types, builtins, pathlib
```

```
import pidgy, pandas, types, builtins, pathlib
```

[2]:
```
pidgy is a literate program and it uses a mix of files for its source.  Until, the␣
↪tangle step
of literate programming we have to rely on available files with avaiable improters,
afterwards markdown is used as source.

{{file_counts.to_html()}}

    file_counts = pandas.Series({k:pathlib.Path(v.__file__).suffix for k,v in␣
↪vars(pidgy).items() if isinstance(v, types.ModuleType) and v is not builtins}).
↪value_counts().to_frame('extensions').T
```

pidgy is a literate program and it uses a mix of files for its source. Until, the tangle step of literate programming we have to rely on available files with avaiable improters, afterwards markdown is used as source.

```
<tr style="text-align: right;">
  <th></th>
  <th>.md</th>
  <th>.ipynb</th>
  <th>.py</th>
</tr>
```

```
<tr>
  <th>extensions</th>
  <td>7</td>
  <td>3</td>
  <td>2</td>
</tr>
```

```
file_counts = pandas.Series({k:pathlib.Path(v.__file__).suffix for k,v in vars(pidgy).
↪items() if isinstance(v, types.ModuleType) and v is not builtins}).value_counts().
↪to_frame('extensions').T
```

[3]:
```
## Shebangs

Perhaps one of the more damning shortcomings of the notebook is that it is no a␣
↪script, and requires specialized software to execute.

`pidgy` markdown documents can begin with a shebang line that defines how the
script should be executed.

```bash
#!/usr/bin/env python -m pidgy run file.md
```

When the shebang is included the markdown file can be executed at the command line␣
↪with a preceeding period.

```bash
./file.md
```
```

Shebangs

Perhaps one of the more damning shortcomings of the notebook is that it is no a script, and requires specialized software to execute.

`pidgy` markdown documents can begin with a shebang line that defines how the script should be executed.

```
#!/usr/bin/env python -m pidgy run file.md
```

When the shebang is included the markdown file can be executed at the command line with a preceeding period.

```
./file.md
```

[4]:
```
## Outcomes
```

(continues on next page)

(continued from previous page)

```
There are numerous outcomes of a source written literate code.  With `pidgy`
as an example we can do more than simply tangle and weave a program.
```

Outcomes

There are numerous outcomes of a source written literate code. With `pidgy` as an example we can do more than simply tangle and weave a program.

[5]:
```
## Best practices for literate programming in Markdown

    import json
Our core moral commitment is to write literate programs, because:

> ...; surely nobody wants to admit writing an illiterate program.
>
> > - [Donald Knuth] _[Literate Programming]_

- Restart and run all or it didn't happen.

  A document should be literate in all readable, reproducible, and reusable
  contexts.

* [Markdown] documents are sufficient for literate progr.

  Markdown documents that translate to python can encode literate programs in a
  form that is better if version control systems that the `json` format that
  encodes notebooks.

* All code should compute.

  Testing code in a narrative provides supplemental meaning to the `"code"`
  signifiers. They provide a test of veracity at least for the computational
  literacy.

* [readme.md] is a good default name for a program.

  Eventually authors will compose [readme.md] documents that act as both the
  `"__init__"` method and `"__main__"` methods of the program.

* Each document should stand alone,
  [despite all possibilities to fall.](http://ing.univaq.it/continenza/Corso%20di
↪%20Disegno%20dell'Architettura%202/TESTI%20D'AUTORE/Paul-klee-Pedagogical-
↪Sketchbook.pdf#page=6)
* Use code, data, and visualization to fill the voids of natural language.
* Find pleasure in writing.

* When in doubt, abide [Web Content Accessibility Guidelines][wcag] so that
  information can be accessed by differently abled audiences.

* First markdown cell is the docstring

[wcag]: https://www.w3.org/WAI/standards-guidelines/wcag/
[donald knuth]: #
[literate programming]: #
```

(continues on next page)

(continued from previous page)

```
[markdown]: #
[readme.md]: #
```

Best practices for literate programming in Markdown

```
import json
```

Our core moral commitment is to write literate programs, because:

> . . . ; surely nobody wants to admit writing an illiterate program.
>
> - *Donald Knuth 'Literate Programming <#>'__*

- Restart and run all or it didn't happen.

A document should be literate in all readable, reproducible, and reusable contexts.

- *Markdown* documents are sufficient for literate progr.

Markdown documents that translate to python can encode literate programs in a form that is better if version control systems that the `json` format that encodes notebooks.

- All code should compute.

Testing code in a narrative provides supplemental meaning to the `"code"` signifiers. They provide a test of veracity at least for the computational literacy.

- *readme.md* is a good default name for a program.

Eventually authors will compose *readme.md* documents that act as both the `"__init__"` method and `"__main__"` methods of the program.

- Each document should stand alone, despite all possibilities to fall.
- Use code, data, and visualization to fill the voids of natural language.
- Find pleasure in writing.
- When in doubt, abide Web Content Accessibility Guidelines so that information can be accessed by differently abled audiences.
- First markdown cell is the docstring

[6]:
```
## External tools

`pidgy` works jupyter notebook, jupyterlab, nteract, and colab.
```

External tools

`pidgy` works jupyter notebook, jupyterlab, nteract, and colab.

[7]:
```
# Conclusion

Markdown documents represent a compact form of a literate programming.
```

Conclusion

Markdown documents represent a compact form of a literate programming.

CHAPTER 3

pidgy tests

3.1 Test `pidgy.tangle`

[1]:
```
        import pidgy, ast
```

[2]:
```
    dir(pidgy.tangle)
```

[2]:
```
['BlockLexer',
 'ExtraSyntax',
 'IPython',
 'Tangle',
 '__all__',
 '__builtins__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '__test__',
 'ast',
 'demojize',
 'init_json',
 'mistune',
 'normalizer',
 'pidgy',
 'pidgyManager',
 'tangle',
 'typing']
```

[3]:
```
    tangle = pidgy.tangle.Tangle()
```

[4]:
```
    s = """---
    a: front matter
    ---

    This is a paragraph.

    * a list

        def f():

    A docstring

            print

    """
```

Unnormalized tokens.

[5]:
```
    tangle.parse(s, normalize=False)
```

[5]:
```
[{'type': 'front_matter', 'text': '\na: front matter\n'},
 {'type': 'paragraph', 'text': 'This is a paragraph.'},
 {'type': 'list_start', 'ordered': False},
 {'type': 'loose_item_start'},
 {'type': 'text', 'text': 'a list'},
 {'type': 'newline'},
 {'type': 'text', 'text': '  def f():'},
 {'type': 'list_item_end'},
 {'type': 'list_end'},
 {'type': 'paragraph', 'text': 'A docstring'},
 {'type': 'code', 'lang': None, 'text': '    print'}]
```

Normalized block tokens

[6]:
```
    tangle.parse(s, normalize=True)
```

[6]:
```
[{'type': 'front_matter', 'text': '\na: front matter\n'},
 {'type': 'paragraph',
  'text': '\nThis is a paragraph.\n\n* a list\n\n    def f():\n    \nA docstring'},
 {'type': 'code', 'lang': None, 'text': '\n    \n        print'}]
```

Normalized block tokens

[7]:
```
    print(tangle.stringify(tangle.parse(s)))
```

```
        locals().update(__import__('ruamel.yaml').yaml.safe_load(
        """a: front matter"""
        ))

        """This is a paragraph.

        * a list

            def f():

        A docstring"""

        print
```

[8]:
```
    transform = pidgy.tangle.pidgyManager().transform_cell
```

[9]:
```
    print(transform(s))
```

```
locals().update(__import__('ruamel.yaml').yaml.safe_load(
"""a: front matter"""
))

"""This is a paragraph.

* a list

    def f():

A docstring"""

print
```

[10]:
```
    print(pidgy.tangle.demojize("""
        = 10
    """))
```

```
    _robot_face__panda_face_  = 10
```

[11]:
```
    ast.parse(transform("""
        return 100
    """)).body
```

[11]:
```
[<_ast.Return at 0x10b3afeb8>]
```

[12]:
```
    pidgy.tangle.ExtraSyntax().visit(ast.parse(transform("""
        return 100
    """))).body[0].value
```

[12]:
```
<_ast.Call at 0x109043080>
```

3.2 Test `pidgy.runpidgy`

[1]:
```
    import pidgy
```

```
import pidgy
```

[2]:
```
    >>> dir(pidgy.runpidgy)
    ['Runner', '__all__', '__builtins__', '__doc__', '__file__', '__loader__', '__
↪name__', '__package__', '__spec__', '__test__', 'ast', 'importlib', 'pidgy', 'render
↪', 'run', 'runpy', 'types']
```

```
>>> dir(pidgy.runpidgy)
['Runner', '__all__', '__builtins__', '__doc__', '__file__', '__loader__', '__name__',
↪ '__package__', '__spec__', '__test__', 'ast', 'importlib', 'pidgy', 'render', 'run
↪', 'runpy', 'types']
```

[3]:
```
    foo = None
```

```
foo = None
```

[4]:
```
The `"templated_document.md"` file has `foo` annotated meaning that it will be␣
↪recognized as CLI parameter.
```

The "templated_document.md" file has foo annotated meaning that it will be recognized as CLI parameter.

[5]:
```
    >>> pidgy.runpidgy.run('templated_document.md')
    __main__
    <module 'templated_document.md' from 'templated_document.md'>
```

```
>>> pidgy.runpidgy.run('templated_document.md')
__main__
<module 'templated_document.md' from 'templated_document.md'>
```

[6]:
```
    >>> pidgy.runpidgy.render('templated_document.md')
    __main__
    "\n...foo... is defined as 42\n\nMy document recieved ['...__main__.py', 'kernel',
↪ 'start', '-f', '...'] as arguments.\n"
```

```
>>> pidgy.runpidgy.render('templated_document.md')
__main__
"\n...foo... is defined as 42\n\nMy document recieved ['...__main__.py', 'kernel',
↪'start', '-f', '...'] as arguments.\n"
```

[7]:
```
    >>> pidgy.runpidgy.render('templated_document.md', foo=900)
    __main__
    "\n...foo... is defined as 900\n\nMy document recieved ['...__main__.py', 'kernel
↪', 'start', '-f', '...'] as arguments.\n"
```

```
>>> pidgy.runpidgy.render('templated_document.md', foo=900)
__main__
"\n...foo... is defined as 900\n\nMy document recieved ['...__main__.py', 'kernel',
↪'start', '-f', '...'] as arguments.\n"
```

[8]:
```
    runner = pidgy.runpidgy.Runner('templated_document.md')
```

```
runner = pidgy.runpidgy.Runner('templated_document.md')
```

[9]:
```
    >>> runner.render(foo=900)
    __main__
    "\n...foo... is defined as 900\n\nMy document recieved ['...__main__.py', 'kernel
↪', 'start', '-f', '...'] as arguments.\n"
```

```
>>> runner.render(foo=900)
__main__
"\n...foo... is defined as 900\n\nMy document recieved ['...__main__.py', 'kernel',
↪'start', '-f', '...'] as arguments.\n"
```

[10]:
```
    >>> runner.run()
    __main__
```

(continues on next page)

(continued from previous page)

```
    <module 'templated_document.md' from 'templated_document.md'>
```

```
>>> runner.run()
__main__
<module 'templated_document.md' from 'templated_document.md'>
```

[11]:
```
    >>> with pidgy.util.argv('script --foo 900'):
    ...     try: runner.cli()
    ...     except SystemExit: None
    __main__
    ...foo... is defined as 900
    <BLANKLINE>
    My document recieved ['script', '--foo', '900'] as arguments.
    <BLANKLINE>
```

```
>>> with pidgy.util.argv('script --foo 900'):
...     try: runner.cli()
...     except SystemExit: None
__main__
...foo... is defined as 900

My document recieved ['script', '--foo', '900'] as arguments.
```

[ ]:
```
```

3.3 Test `pidgy.magic`

pidgy automatically provides the magics when it is imported interactively.

```
import pidgy
from IPython import get_ipython
```

```
%%tangle
This is my pidgy

    print("This is my code")
```

```
%%tangle --tokens
This is my pidgy

    print("This is my code")
```

```
foo = 900
```

```
%%render
I am a Markdown template that can display {{foo}}
```

[1]:
```
import pidgy
from IPython import get_ipython
ip = get_ipython()
```

```
import pidgy
from IPython import get_ipython
ip = get_ipython()
```

[2]:
```
if ip:
    !pidgy
```

```
Usage: pidgy [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  kernel
  run       `pidgy` `run` executes `pidgy` documents as programs.
  template  `pidgy` `template` executes `pidgy` documents as programs and...
  test      Formally test markdown documents, notebooks, and python files.
  to        Convert pidgy documents to other formats.
```

```
if ip:
    !pidgy
```

[3]:
```
if ip:
    !pidgy run --help
```

```
Usage: pidgy run [OPTIONS] REF

  `pidgy` `run` executes `pidgy` documents as programs.

Options:
  --help  Show this message and exit.
```

```
if ip:
    !pidgy run --help
```

[4]:
```
if ip:
    !pidgy test --help
```

```
Usage: pidgy test [OPTIONS] [FILES]...

  Formally test markdown documents, notebooks, and python files.

Options:
  --help  Show this message and exit.
```

```
if ip:
    !pidgy test --help
```

[5]:
```
if ip:
    !pidgy kernel --help
```

```
Usage: pidgy kernel [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  install    `install` the pidgy kernel.
  start      Launch a `pidgy` kernel applications.
  uninstall  `uninstall` the kernel.
```

```
if ip:
    !pidgy kernel --help
```

[6]:
```
    if ip:
        !pidgy run templated_document.md
```

```
Running templated_document.md.
__main__
```

```
if ip:
    !pidgy run templated_document.md
```

[7]:
```
    if ip:
        !pidgy template templated_document.md
```

```
__main__
`foo` is defined as 42

My document recieved ['templated_document.md'] as arguments.
```

```
if ip:
    !pidgy template templated_document.md
```

[8]:
```
    # NBVAL_SKIP
    if ip:
        !chmod u+x templated_document.md
        !./templated_document.md
```

```
__main__
`foo` is defined as 42

My document recieved ['./templated_document.md'] as arguments.
```

```
# NBVAL_SKIP
if ip:
    !chmod u+x templated_document.md
    !./templated_document.md
```

3.4 Test 3rd party integrations.

[ ]:
```
import papermill, pathlib, tqdm
```

[1]:
```
def remove_output(path):
    try: __import__('os').remove(path)
    except: ...
```

[2]:
```
def test_execute():
    in_, out = pathlib.Path('pidgy/tests/parameterized_notebook.md.ipynb'), pathlib.
↪Path('pidgy/tests/_parameterized_notebook.md.ipynb')
    assert in_.exists()
    papermill.execute_notebook(str(in_), str(out), kernel_name='pidgy',␣
↪parameters=dict(alpha=100))
    assert out.exists()
    remove_output(out)
```

3.5 Create a simple `fastapi` application

fastapi is a web application framework in [Python]. It uses type annotations to define endspoints.

```
import fastapi, click
```

Make an instance of a `fastapi` application.

```
app = fastapi.FastAPI()


@app.get('/highlight/{str}')
def highlight_terminal(str:str):


    import pygments.formatters.terminal256
    return pygments.highlight(str, pygments.lexers.find_lexer_class_by_name('yaml')(),
↪ pygments.formatters.terminal256.Terminal256Formatter(style='bw'))

@app.get('/upper/{str}')
def upper(str:str):
```

`upper` returns the uppercase value of the input string.

return str.upper()

@click.group() def cli(): . . .

@cli.command() def schema():

Display the `schema` for our simple application.

click.echo(highlight_terminal(**import**('yaml').safe_dump(app.openapi(), default_flow_style=False)))

@cli.command() def serve():

Serve the simple `fastapi` application.

import('uvicorn').run(app, host="0.0.0.0", port=8000)

```
__name__ == "__main__" and cli()

def _test_app():
    __import__('nest_asyncio').apply()
    import starlette.testclient
    client = starlette.testclient.TestClient(app)
    assert client.get('/upper/rawr').text == 'RAWR!'
```

3.6 Tidy Data

tidy data

[2]:
```
> [...], a stack of elements is a common abstract data type used in computing. We␣
↪would not think 'to add' two stacks as we would two integers.
>> Jeanette Wing - [Computational thinking and thinking about␣
↪computing][computational thinking]


[computational thinking]: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2696102/
```

> [...], a stack of elements is a common abstract data type used in computing. We would not think 'to add' two stacks as we would two integers. > Jeanette Wing - Computational thinking and thinking about computing

[3]:
```
A modernist style of notebook programming persists where documents are written as if␣
↪programs are
starting for nothing. Meanwhile, authors of R programming language tend to begin with␣
↪the assumption
that data exists and so does code. Notebook are a powerful substrate for working with␣
↪data and
describing the logic behind different permutations.

pidgy was designed to weave projections of tabular into a computational documentation.
↪ Specifically,
we are concerned with the DataFrame, a popular tidy data abstraction that serves as a␣
↪first
class data structure in scientific computing.
```

A modernist style of notebook programming persists where documents are written as if programs are starting for nothing. Meanwhile, authors of R programming language tend to begin with the assumption that data exists and so does code. Notebook are a powerful substrate for working with data and describing the logic behind different permutations.

pidgy was designed to weave projections of tabular into a computational documentation. Specifically, we are concerned with the DataFrame, a popular tidy data abstraction that serves as a first class data structure in scientific computing.

[4]:
```
    import pandas as
```

```
import pandas as
```

[5]:
```
<hr/>
```

[6]:
```
The figure above illustrates the information in `df`.

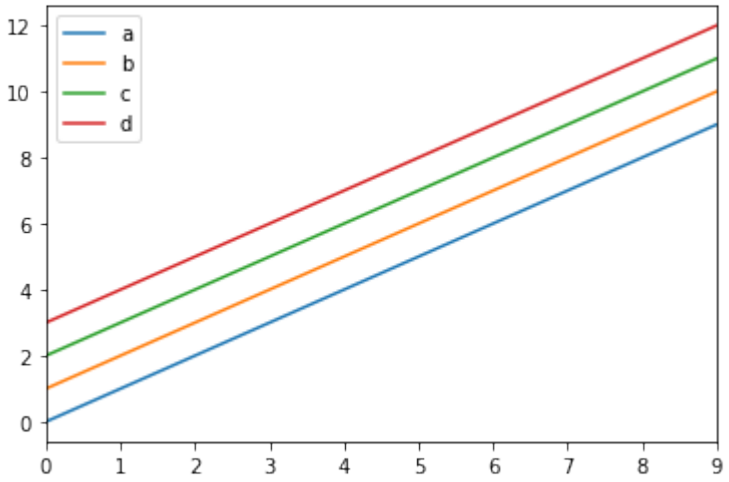
A high level numeric project of this data's statistics are:

{{df.describe().to_html()}}

The statistics were created using measurements that look like the following data:

{{df.head(2).to_html()}}

    df = .DataFrame([range(i, i+4) for i in range(10)], columns=list('abcd'))
    df.plot();
```



The figure above illustrates the information in `df`.

A high level numeric project of this data's statistics are:

```
<tr style="text-align: right;">
  <th></th>
  <th>a</th>
  <th>b</th>
  <th>c</th>
  <th>d</th>
</tr>
```

(continues on next page)

(continued from previous page)

```
<tr>
  <th>count</th>
  <td>10.00000</td>
  <td>10.00000</td>
  <td>10.00000</td>
  <td>10.00000</td>
</tr>
<tr>
  <th>mean</th>
  <td>4.50000</td>
  <td>5.50000</td>
  <td>6.50000</td>
  <td>7.50000</td>
</tr>
<tr>
  <th>std</th>
  <td>3.02765</td>
  <td>3.02765</td>
  <td>3.02765</td>
  <td>3.02765</td>
</tr>
<tr>
  <th>min</th>
  <td>0.00000</td>
  <td>1.00000</td>
  <td>2.00000</td>
  <td>3.00000</td>
</tr>
<tr>
  <th>25%</th>
  <td>2.25000</td>
  <td>3.25000</td>
  <td>4.25000</td>
  <td>5.25000</td>
</tr>
<tr>
  <th>50%</th>
  <td>4.50000</td>
  <td>5.50000</td>
  <td>6.50000</td>
  <td>7.50000</td>
</tr>
<tr>
  <th>75%</th>
  <td>6.75000</td>
  <td>7.75000</td>
  <td>8.75000</td>
  <td>9.75000</td>
</tr>
<tr>
  <th>max</th>
  <td>9.00000</td>
  <td>10.00000</td>
  <td>11.00000</td>
  <td>12.00000</td>
</tr>
```

(continues on next page)

(continued from previous page)

The statistics were created using measurements that look like the following data:

```
<tr style="text-align: right;">
  <th></th>
  <th>a</th>
  <th>b</th>
  <th>c</th>
  <th>d</th>
</tr>
```

```
<tr>
  <th>0</th>
  <td>0</td>
  <td>1</td>
  <td>2</td>
  <td>3</td>
</tr>
<tr>
  <th>1</th>
  <td>1</td>
  <td>2</td>
  <td>3</td>
  <td>4</td>
</tr>
```

```
df = .DataFrame([range(i, i+4) for i in range(10)], columns=list('abcd'))
df.plot();
```

[7]:
```
<hr/>
```

[8]:
```
In technical writing we need to consider existing conventions like:
* Figures above captions
* Table below captions

It still remains to be seen where code canonically fits in reference to figures and␣
↪tables.

[Why should a table caption be placed above the table?]

[Why should a table caption be placed above the table?]: https://tex.stackexchange.
↪com/questions/3243/why-should-a-table-caption-be-placed-above-the-table
```

In technical writing we need to consider existing conventions like: * Figures above captions * Table below captions

It still remains to be seen where code canonically fits in reference to figures and tables.

Why should a table caption be placed above the table?

[9]:
```
[notebook war]

[notebook war]: https://yihui.org/en/2018/09/notebook-war/
```

notebook war

3.7 Figures

[1]:
```
    from graphviz import Source
    Ø = __name__ == '__main__'
```

```
from graphviz import Source
Ø = __name__ == '__main__'
```

[12]:
```
<!--

    tangle_weave = Source(
digraph {rankdir=LR
subgraph cluster_pidgy {
    label="PIDGY literate programming"
    MD->MARKDOWN[label=WEAVE]
    MARKDOWN->HTML[label=PANDOC style=dashed]
    HTML->PDF[label=PRINT style=dashed]
    MD->IPY[label=TANGLE]
    IPY->PYC[label=PYTHON]
    IPYNB->IPY[label=TANGLE style=dotted]
    IPYNB->MARKDOWN[label=WEAVE]
}
subgraph cluster_knuth {
    label="WEB literate programming"
    WEB->TEX[label=WEAVE] TEX->DVI[label=TEX]
    WEB->PAS[label=TANGLE] PAS->REL[label=PASCAL]
}
}

    , filename='tange_weave.png')
    tangle_weave.save()
    if Ø: return tangle_weave

-->

    tangle_weave:\
The diagram of tangling and weaving compares the original literate programming
approach to the markdown-forward approach.  The weaving
step becomes an identity transformation.  Tangling
markdown to python is a line-by-line, to maintain proper assertions,
transformation that requires wrapping `not "code"` in quotations and indenting them␣
↪properly.
IPY is a superset of python provided by IPython and includes shell, magics, and macro␣
↪commands.
```

```
tangle_weave:\
```

The diagram of tangling and weaving compares the original literate programming approach to the markdown-forward approach. The weaving step becomes an identity transformation. Tangling markdown to python is a line-by-line, to maintain proper assertions, transformation that requires wrapping `not "code"` in quotations and indenting them properly. IPY is a superset of python provided by IPython and includes shell, magics, and macro commands.

CHAPTER 4

Source

4.1 API Reference

This page contains auto-generated API reference documentation[1].

4.1.1 pidgy

Subpackages

pidgy.pytest_config

Package Contents

pidgy.pytest_config.**pytest_collect_file**(*parent*, *path*)

Submodules

pidgy.__main__

pidgy.autocli

Module Contents

pidgy.autocli.**autoclick**(**object: typing.Union[types.FunctionType, click.Command], group=None, **settings*) → click.Command

Automatically generate a click command line application using type inference.

pidgy.autocli.**istype**(*x: typing.Any*, *y: type*) → bool

[1] Created with sphinx-autoapi

pidgy.autocli.**click_type**(*object: typing.Union[type, tuple], default=None*) → typing.Union[type, click.types.ParamType]
Translate python types to click's subset of types.

pidgy.autocli.**command_from_decorators**(*command*, **decorators*, ***settings*)

pidgy.autocli.**decorators_from_dicts**(*annotations*, *defaults*, **decorators*)

pidgy.autocli.**decorators_from_dict**(*object*)

pidgy.autocli.**decorators_from_module**(*object*)

pidgy.autocli.**signature_to_decorators**(*object*, **decorators*)

pidgy.base

Module Contents

pidgy.base.**implementation**

pidgy.base.**specification**

pidgy.base.**plugin_manager**

pidgy.util

Module Contents

class pidgy.util.**ContextDepth**
Count the current depth of a context manager invocation.

depth = 0

__enter__(*self*)

__exit__(*self*, **e*)

pidgy.util.**html_comment**(*text*)

pidgy.util.**istype**(*object*, *cls*)

pidgy.util.**WHITESPACE**

pidgy.util.**indents**(*str: str*) → int

pidgy.util.**num_first_indent**(*text: str*) → int
The number of indents for the first blank line.

pidgy.util.**num_last_indent**(*text: str*) → int
The number of indents for the last blank line.

pidgy.util.**base_indent**(*tokens: typing.List[dict]*) → int
Peek into mistune tokens and find the last code indent.

pidgy.util.**quote**(*text: str*) → str
Wrap strings in triple quoted block strings.

pidgy.util.**whiten**(*text: str*) → str
whiten strips empty lines because the *markdown.BlockLexer* doesn't like that.

pidgy.util.**strip_front_matter**(*text: str*, *sep=None*) → str
Remove yaml front matter froma string.

pidgy.util.**ansify**(*str: str*, *format='md'*)
High source to be printed in the terms.

pidgy.util.**yield_files**(*files: typing.Sequence[str], recursive=False*) → typing.Generator
Return a list of files from a collection of files and globs.

pidgy.util.**argv**(**args: str*)

pidgy.util.**strip_shebang**(*str*)

pidgy.util.**strip_html_comment**(*str*)

pidgy.util.**strip_front_matter**(*text: str*, *sep=None*) → str
Remove yaml front matter froma string.

pidgy.util.**sys_path**()

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

H

I

N

P

Q

S

W

Y