Python

deathbeds

Sep 12, 2020

LITERATE SHELL

`pidgy` is a fun way to interactively program in Markdown and `IPython`. It is design to tell stories with code, tests, and data in your favorite IDE (jupyter, nteract, colab, vscode). It that allows fluid combinations of code and prose with added language features like block markdown variables, emoji variables names, and interactive formal testing. It is designed primarily for Jupyter notebooks and Markdown source files that can be used as python modules, scripts, and applications.

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

CHAPTER

ONE

PIDGY FEATURES

- interleave narrative and code in the same cells
- test literate notebooks and programs
- transclude real data into narratives with `jinja2` templates
- reactive displays that update on rendering

CHAPTER

TWO

THE PIDGY SHELL/KERNEL

`pidgy` is installed as `jupyter` kernel that can be used in lab or classic. `pidgy` opens authors into a markdown forward programming interface.

- the kernel can be installed manually using the cli.

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

CHAPTER

THREE

AUTHORING PIDGY DOCUMENTS

in pidgy, code is indented. both markdown and python cells accept markdown in pidgy. as a result, in pidgy markdown cells are consider off and code are considered on. the indented code pattern is valid in standard IPython kernels and pidgy.

CHAPTER

FOUR

IMPORTING `PIDGY` DOCUMENTS

after computing your `pidgy` programs you may reuse them as modules. `pidgy` extends the python import system to include `".ipynb"` and `".md"` files along with native `".py"` files.

```
with __import__("pidgy").pidgyLoader():
    import README
```

CHAPTER

FIVE

THE `PIDGY` CLI

the `pidgy` cli helps to tangle and weave entire literate pidgy programs.

```
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

SIX

DEVELOPER

`pidgy` uses `doit` to make tests and documentation work.

```
import doit.tools
```

6.1 tasks

```
def task_book():
```

`pidgy` builds document with the `jupyter_book` project.

```
    return dict(actions="jupyter-book build .".splitlines())

def task_sphinx():
```

the `"conf.py"` for sphinx in generated from the `jupyter_book` cli and built with `sphinx`.

```
    return dict(actions="sphinx-build . docs".splitlines())

def task_test():
```

`pidgy` tests notebooks using plugins from `importnb` and `nbval`

```
    return dict(actions=[
        doit.tools.Interactive("pytest --nbval --sanitize-with sanitize.cfg -p␣
↪no:warnings pidgy/tests/test_* docs/examples")
    ])
```

6.1.1 Discussion

```
    import pidgy, jupyter
```

```
import pidgy, jupyter
```

```
## Markdown for literate programming

`pidgy` uses the Markdown formatting language because it is the native document format
for `jupyter`. Another motivation is the broad application of Markdown for literate
progamming in different programming languages. Markdown could potentially provide a
```

(continues on next page)

(continued from previous page)

```
generalized document for polyglot literate computing.

We find are two brands of Markdown programming implementations:

1. Those that use indented code blocks.
2. Those that use code fences.

`pidgy` follows the influence of literate coffeescript and focuses on the ability to␣
↪tangle
code primarily written using indented code fences. It does accomodate code fences, but
currently `pidgy` holds no opinion about how to treat code fence syntax like in␣
↪RMarkdown and PWeave.
```

Non-consecutive header level increase; 0 to 2

```
## Alternative source files

    import wtypes, pandas, pathlib, types, builtins

When code is written primarily in source files, the literary quality of the program is
always secondary to the computational qualities. `pidgy` includes the ability to use
alternative file schema to import literate documents as python modules, tests, and␣
↪software.
The reusuability of `pidgy` programs means that each document can have multiple␣
↪objectives of literary and computational quality.

`pidgy` is an implementation that follows many other experiements in interactive␣
↪computing with notebooks.
`importnb` is a primary library of `pidgy` that customizes Python's import hooks.
When software begins from literate documents like notebooks and markdown files it can␣
↪then mature
to scripts as each idea materializes. `importnb` and `wtypes` represent two projects␣
↪that
mature from notebook source files that are idea for fluid ideas to python scripts with
literate programs for tests and documentation.

`pidgy` 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.
Each file type has different capacities for encoding literate narratives.

* Python scripts favor language, but doctests represent literate source code for␣
↪testing.
* Markdown scripts capture the full input for a literate program.
* Notebooks are hypermedia collages of literate programs that connect literate␣
↪computing input to output.

{{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
```

Non-consecutive header level increase; 0 to 2

```
## Shebangs

```

(continues on next page)

(continued from previous page)

```
    import json

Perhaps one of the more damning shortcomings of the notebook is that it is not a␣
↪script,
and requires specialized software to execute. Notebooks are encoded in `json` and␣
↪cannot
include command lines. As a result, they cannot include a shebang line. One the other␣
↪hand,
`pidgy` markdown files can include a shebang line that describes how a literate␣
↪program
should tangle. `pidgy` has 3 options, run template test, should be tangle weave


An example `"file.md"` could begin with one of the following shebang statements

```bash
#!/usr/bin/env python -m pidgy run
#!/usr/bin/env python -m pidgy template
#!/usr/bin/env python -m pidgy test
```

Then the markdown file can be executed at the command line with a preceeding period.

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

Non-consecutive header level increase; 0 to 2

```
## Revision Control

Another struggle with notebooks is revision control, git is not the solution.␣
↪Notebooks are data
and require a different strategy than git. Markdown documents on the other hand
capture the input of literate programming and diff really well.
```

```
## 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.
```

Non-consecutive header level increase; 0 to 2

```
## Best practices for literate programming in Markdown


* Programs should be literate.
* Restart and run all or it didn't happen.

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


* Start with natural language and a docstring.

  Most notebooks begin importing code, literate programs should begin importing
```

(continues on next page)

(continued from previous page)

```
  natural language to describe a goal.

* 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.
* 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.


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

Non-consecutive header level increase; 0 to 2

```
## External tools

Since `pidgy` relies on `jupyter`s open standards it works with jupyter notebook,
jupyterlab, nteract, and colab. We were unable to spend creating plugins for Atom or␣
↪VSCode.
```

Non-consecutive header level increase; 0 to 2

```
# Conclusion


Notebooks and Markdown files are substrates for literature backed by programming␣
↪languages.
Choosing to write programs as literature can drastically improve the quality of␣
↪program
that would otherwise rely on the programming language syntax. Authoring `pidgy`␣
↪documents is
a fun experience that allows the author vascillate between writing code and narrative.
↪ The ability
to transclude interactive computation into documents gives authors the ability to␣
↪access
new language and meaning when supplemented with code.
```

6.1.2 the `pidgy` shell

```
import ipykernel.kernelapp, ipykernel.zmqshell, traitlets, pidgy, types, IPython,␣
↪jinja2
```

`pidgy` relies on the `jupyter` shell & kernel to provide an enhanced authoring experience for computational documentations. the kernel has the ability to work with the cpu, memory, and other devices; the shell is the application we use to affect changes to the kernel.

the `jupyter` ecosystem has grown include more over 100 languages now. Most recently, it has brought new life to compiled languages like C++ and Fortran.

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

```
class pidgyShell(ipykernel.zmqshell.ZMQInteractiveShell):
```

`pidgy` introduces markdown as an application language for literate computing. it encodes the tangle and weave aspects of literate programming directly into the interactive cell execution. with each execution, the input is *tangled* into valid `IPython` source code then the input is processed as [markdown] output with `jinja2` templates.

tangle markdown to code

```
    @traitlets.default('input_transformer_manager')
    def _default_tangle(self):
```

`pidgy` tangles [markdown] to source code on block elements permitting line-for-line transforms to `IPython`. The *tangle* section contains more detail on this heurisitic.

```
        return pidgy.tangle.pidgyManager()
```

we can already use existing `IPython` features to transform the abstract syntax tree and apply text processing. `pidgy` includes the abilities to:

- use emojis in code

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

- use return statements at the top code level, outside of functions, to display objects.

  ```
  ast_transformers = traitlets.List([pidgy.tangle.ExtraSyntax()])
  ```

weave input to output

```
    weave = traitlets.Any()
    @traitlets.default('weave')
    def _default_weave(self):
```

`pidgy` weaves the input into a rich display provided by `jupyter` display system, it adds the ability to implicitly transclude variables from live compute into a narrative. in traditional literate computing, code and narrative had to be mixed explicitly.

```
        return pidgy.weave.Weave(parent=self)
```

formal interactive testing

```
    testing = traitlets.Any()
    @traitlets.default('testing')
    def _default_testing(self):
```

`pidgy` promotes value by including formal testing it literate programs.

```
        testing = pidgy.testing.Testing(parent=self)
        self.ast_transformers.append(testing.visitor)
        return testing
```

import alternative document formats

```
    loaders = traitlets.Dict()

    def init_loaders(self):
```

`pidgy` augments the python import system to include `pidgy` notebooks and markdown documents along with normal notebooks.

```
        for x in (pidgy.pidgyLoader, __import__("importnb").Notebook):
            if x not in self.loaders:
                self.loaders[x] = x().__enter__()
```

extra shell initialization options

```
    def init_pidgy(self):
        if self.weave is None:
            self.weave = pidgyShell._default_weave(self)
        if self.testing is None:
            self.testing = pidgyShell._default_testing(self)


        for x in (self.weave, self.testing):
            try: x.register()
            except AssertionError:...

        pidgyShell.init_loaders(self)
        pidgy.magic.load_ipython_extension(self)
        __import__('importlib').reload(__import__('doctest'))

    enable_html_pager = traitlets.Bool(True)

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

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

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

6.1.3 tangling/translating code

tangling code, as presented by donald knuth, converts a document language into a programming language. the original implementation converts `".WEB"` files to valid pascal - `".PAS"` - files. the `pidgy` approach begins with [markdown] text that converts to `IPython`. [Markdown]: # [Python]: #

```
    import typing, IPython, pidgy.util, ast, textwrap, markdown_it
```

tangling `pidgy` uses block level lexical analysis to separate non-code and code lines of code in an input; `pidgy` does not take any opinion on inline level markdown syntax. the `PythonRender` uses the `markdown_it` module for parsing markdown; past versions of `pidgy` have tried `#pandoc, mistune, and mistletoe`. `markdown_it` is the preferred parser because it provides line numbers for markdown tokens.

```
    class Tangle(pidgy.compat.markdown.Markdown):
        def __init__(self, *args, **kwargs):
            kwargs['renderer_cls'] = kwargs.get('renderer_cls', PythonRender)
            super().__init__(*args, **kwargs)
            [self.block.ruler.before(
                "code",
                "front_matter",
                __import__('functools').partial(pidgy.util.frontMatter, x),
                {"alt": ["paragraph", "reference", "blockquote", "list"]},
            ) for x in "-+"]
            self.block.ruler.before(
                "reference", "footnote_def", markdown_it.extensions.footnote.index.
↪footnote_def, {"alt": ["paragraph", "reference"]}
            )
            self.disable('html_block')
```

the primary goal of the `pidgy` lexical analysis to separate non-code and code lines when the markdown is *pythonified*. both indented block code and code fences determine the heuristics for entangling the non-code and code strings. while developing `pidgy`, we've purposefully avoided defining any heuristics for code fenced languages. if author's prefer they can executed code in `pidgy` code fences if no language is supplied.

```
    class Pythonify(pidgy.compat.markdown.Renderer):
        QUOTES = '"""', "'''"

        def noncode(self, tokens, idx, env):
            token, range, prior = None, slice(None), slice(*tokens[-1].map)
            if idx < len(tokens):
                token = tokens[idx]
                range, prior = slice(*tokens[idx].map), slice(*tokens[idx-1].map) if␣
↪idx else slice(0,0)

            non_code = pidgy.util.dedent_block(''.join(env['src'][prior.stop:range.
↪start]))
            non_code = self.indent(self.hanging_indent(non_code, env), env)
            if not env.get('quoted', False):
                non_code = self.quote(non_code, trailing=';' if token is None else '')
            return non_code

        def code_block(self, tokens, idx, options, env):
            code = self.noncode(tokens, idx, env) + pidgy.util.quote_docstrings(self.
↪token_to_str(tokens, idx, env))
            return self.update_env(code, tokens, idx, env) or code

        def fence(self, tokens, idx, options, env):
```

(continues on next page)

(continued from previous page)

```
            "We'll only recieve fences without a lang."
            code =  self.noncode(tokens, idx, env) + textwrap.indent(
                pidgy.util.quote_docstrings(pidgy.util.unfence(self.token_to_
↪str(tokens, idx, env))), ' '*4
            )
            return self.update_env(code, tokens, idx, env) or code


        def update_env(self, code, tokens, idx, env):
            next = self.get_next_code_token(tokens, idx)
            env.update(base_indent=pidgy.util.trailing_indent(code))

            extra_indent = 0
            if next:
                extra_indent = max(0, pidgy.util.lead_indent(env['src'][slice(*next.
↪map)]) -env['base_indent'])
            if not extra_indent and code.rstrip().endswith(":"):
                extra_indent += 4
            rstrip = code.rstrip()
            env.update(
                extra_indent=extra_indent,
                continued=rstrip.endswith('\\'),
                quoted=rstrip.rstrip('\\').endswith(self.QUOTES)
            )
```

`pidgy` includes special affordances affordances for common notation like front matter, footnotes as annotations, and bulleted lists.

```
    class PythonRender(Pythonify):
        def front_matter(self, tokens, idx, options, env):
            token, code = tokens[idx], self.token_to_str(tokens, idx, env)
            if token.markup == '+++':
                code = F'''locals().update(__import__('toml').loads("""{code}""".
↪partition('+++')[2].rpartition('+++')[0]))\n'''
            elif token.markup == '---':
                code = F'''locals().update(__import__('ruamel.yaml').yaml.safe_load(""
↪"{code}""".partition('---')[2].rpartition('---')[0]))\n'''
            return self.indent(code, env)


        def reference(self, tokens, idx, options, env, *, re='link_item'):
            token, code = tokens[idx], self.token_to_str(tokens, idx, env)
            if env['quoted']:
                return code

            expr  = "{"+F"""x.group(1): x.group(2).rstrip() for x in __import__('pidgy
↪').util.{re}.finditer({
                self.quote(textwrap.dedent(code), trailing=")}").rstrip()
            }"""
            if not env['continued']:
                expr = """locals()["__annotations__"] = {**%s, **locals().get('__
↪annotations__', {})}"""%expr
            code = self.noncode(tokens, idx, env) + self.indent(expr + "\n", env)
            return code

        def footnote_reference_open(self, tokens, idx, options, env):
```

(continues on next page)

(continued from previous page)

```
            return self.reference(tokens, idx, options, env, re='footnote_item')

        def bullet_list_open(self, tokens, idx, options, env):
            token, code = tokens[idx], self.token_to_str(tokens, idx, env)
            if env['quoted']:
                return code
            if env['continued']:
                return self.indent(
                    (F"""[x.group().rstrip().partition(' ')[2] for x in __import__(
↪'pidgy').util.list_item.finditer(⦃
                        self.quote(textwrap.dedent(code), trailing=')]')
                    }\n"""), env)
            code = self.quote(textwrap.dedent(code), trailing=';')
            code = self.indent(self.hanging_indent(code, env), env)
            return code

        ordered_list_open = bullet_list_open
```

`tangle` is a public function for tangling markdown to python.

```
    def tangle(str:str)->str:
        translate = Tangle()
        return translate.render(''.join(str or []))
```

`pidgy` interfaces with `IPython` as an input transform manager trait.

```
    class pidgyManager(pidgy.base.Trait, IPython.core.inputtransformer2.
↪TransformerManager):
        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)
            self.tangle = Tangle()
        def transform_cell(self, cell):
            if self.enabled:
                cell = self.tangle.render(cell)
            return super(type(self), self).transform_cell(cell)
```

more langauge features

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

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.

```
    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)
```

(continues on next page)

(continued from previous page)

```
        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.

```
    def demojize(lines, delimiters=('_', '_')):
        str = ''.join(lines or [])
        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: ...
        return ''.join(lines)
```

6.1.4 import markdown sources

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`

```
    __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.

```
    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].

```
    def code(self, str):
        for callable in (self.transformer_manager.transform_cell,
                         pidgy.tangle.demojize):
```

(continues on next page)

(continued from previous page)

```
            str = ''.join(callable(''.join(str)))
        return str
```

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

```
    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.

```
    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
```

6.1.5 Woven text

```
import IPython, pidgy.base, traitlets, jinja2
with pidgy.pidgyLoader(lazy=True): import pidgy.compat.templating
class Weave(pidgy.base.Trait):
```

Nominally, since the earliest illuminated manuscripts, text is in with type and form. In [literate programming], the weave step explicitly refers to the act of converting an input source into other media forms.

The original [WEB] implementation models the properties of printed documents using the [TeX] document language. `pidgy` shares the same concerns with the form of the published document; in fact, this workflow produces a [PDF] document using the [ReadTheDocs] open-source service with [DVI], Knuth's original woven target, as an intermediate product[^dvi].

However, prior to printed forms, `pidgy` is concerned with the ability to `Weave` hypertext and hypermedia forms generated by [literate computing] and composing documents in modern web-browsers. `pidgy` uses as a document formatting language; it is chosen because it is the default document language of `jupyter` technologies.

The source code for `pidgy` is always [Markdown], it provides both the design and computation of an input. In [pidgy]

The `Weave` class controls the display of `pidgy` outputs, and it relies on the `Weave.parent` interactive shell.

```
    environment = traitlets.Instance('jinja2.Environment')
    iframe_width = traitlets.Any("100%")
    iframe_height = traitlets.Any("""600""")

    def post_run_cell(self, result):
        if not self.enabled: return
```

The `Weave` step is invoked after a cell or code has been executed.

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

(continues on next page)

(continued from previous page)

```
        if text.startswith(('http:', "https:")):
            lines in text.splitlines()
            if all(x.startswith(('http:', "https:")) for x in lines):
                return IPython.display.display(*(
                    IPython.display.IFrame(x, width=self.iframe_width, height=self.
↪iframe_height)
                    for x in lines
                ))
```

`pidgy` defers from printing the output if the first line is blank.

```
        display = pidgy.compat.templating.MarkdownDisplay(
            body=text, parent=self.parent, template=self.template(text)
        )
        self.display_manager.append(display)
        display.display()
```

Transclusion with `jinja2` templates.

`jinja2` is a convention for notebooks in the `nbconvert` universe. `jinja2` is a popular templating engine that makes it possible to put programmatic objects into text.

```
    render_template = traitlets.Bool(True).tag(description=
```

`Weave.render_template` is a toggle for turning transclusion on and off.

```
    )
```

By default templates are always rendered, but this feature can be turned off.

```
    def template(self, text):
        import builtins, operator
        return self.environment.from_string(text, globals={
                **vars(builtins), **vars(operator),
                **(getattr(self.parent, 'user_ns', {})).get('__annotations__', {}),
                **getattr(self.parent, 'user_ns', {})})

    def render(self, text):
        if not self.render_template: return text
        import builtins, operator
        try:
            return self.template(text).render()
        except BaseException as Exception: self.parent.showtraceback((type(Exception),
↪ Exception, Exception.__traceback__))
        return text

    display_manager = traitlets.Any()

    @traitlets.default('display_manager')
    def _default_display_manager(self):
        manager = pidgy.compat.templating.DisplayManager(parent=self.parent)
        manager.register()
        return manager
```

(continues on next page)

(continued from previous page)

```
    @traitlets.default('environment')
    def _default_environment(self):
```

More information about the default `jinja2` environment may be found in the [compatability module].

```
        return pidgy.compat.templating.environment(self.parent)
```

6.1.6 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.

```
import pidgy.base, traitlets, ast, unittest, IPython, sys
with pidgy.pidgyLoader(lazy=True):
    import pidgy.compat.unittesting, pidgy.compat.typin

class Testing(pidgy.base.Trait, pidgy.compat.unittesting.TestingBase):
    medial_test_definitions = traitlets.List()
    pattern = traitlets.Unicode('test_')
    visitor = traitlets.Instance('ast.NodeTransformer')
    results = traitlets.List()
    trace = traitlets.Any()
    update = traitlets.Bool(True)

    @traitlets.default('trace')
    def _default_trace(self):
        return pidgy.compat.typin.InteractiveTyping(parent=self)

    @traitlets.default('visitor')
    def _default_visitor(self):
        return pidgy.compat.unittesting.Definitions(parent=self)

    def post_run_cell(self, result):
        if not self.enabled: return
        if not (result.error_before_exec or result.error_in_exec):
            tests = []
            while self.medial_test_definitions:
                name = self.medial_test_definitions.pop(0)
                object = self.parent.user_ns.get(name, None)
                if name.startswith(self.pattern) or pidgy.util.istype(object,␣
↪unittest.TestCase):
                    tests.append(object)

            test = pidgy.compat.unittesting.Test(result=result, parent=self.parent,␣
↪vars=True)
            if self.trace.enabled:
                with self.trace: test.test(*tests)
            else: test.test()
            if test.test_result.testsRun:
                IPython.display.display(test)
                self.results = [
```

(continues on next page)

(continued from previous page)

```
                    x for x in self.results if x._display and test._display and (x.
↪_display.display_id != test._display.display_id)
                ] + [test]

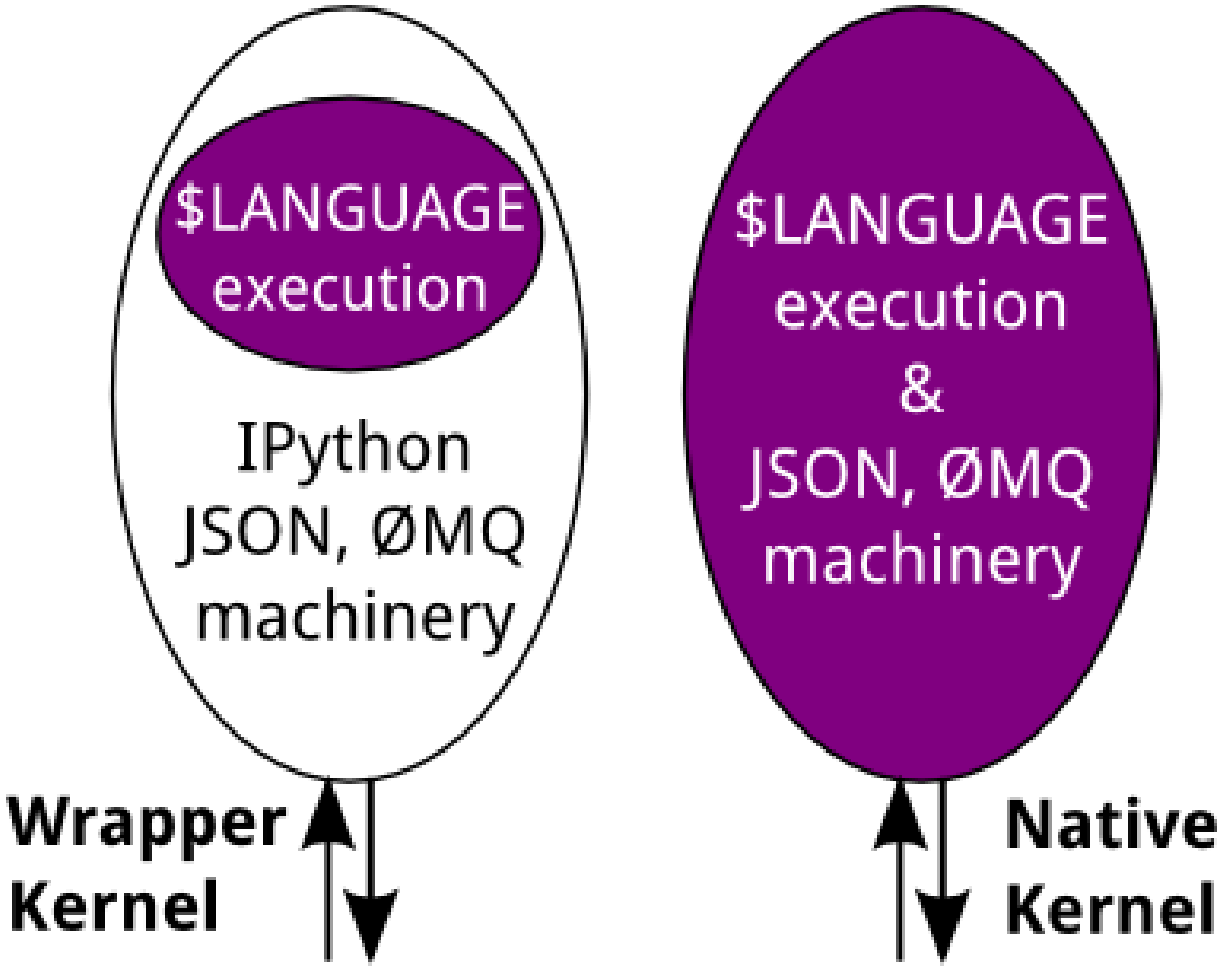

    def stub(self):
        return self.trace.stub()
    def post_execute(self):
        if self.update:
            for test in self.results:
                if self.trace.enabled:
                    with self.trace: test.test()
                else: test.test()
                test.update()
```

6.1.7 `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, object):
```

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

```
        self.shell._last_parent = object
        return super().init_metadata(object)

    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].rstrip()[-3:] == '!!!':
            if code[:cursor_pos].rstrip()[-6:] == '!!!'*2:
                self.shell.run_cell(code[:cursor_pos], silent=True)
            return self.markdown_result(self.shell.weave.render(code[:cursor_pos]))
        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)
...
```

6.1.8 Tidy Data

tidy data

```
    __import__('IPython').core.interactiveshell._should_be_async = lambda x:  False
```

```
__import__('IPython').core.interactiveshell._should_be_async = lambda x:  False
```

```
> ... 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/

        __annotations__
```

```
{'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

```
    __annotations__
```

```
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.

```
    import pandas as
```

```
<module 'pandas' from '/home/tonyfast/miniconda3/lib/python3.7/site-packages/pandas/__
↪init__.py'>
```

```
import pandas as
```

```
    %matplotlib inline
The figure above illustrates the information in `df`.

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

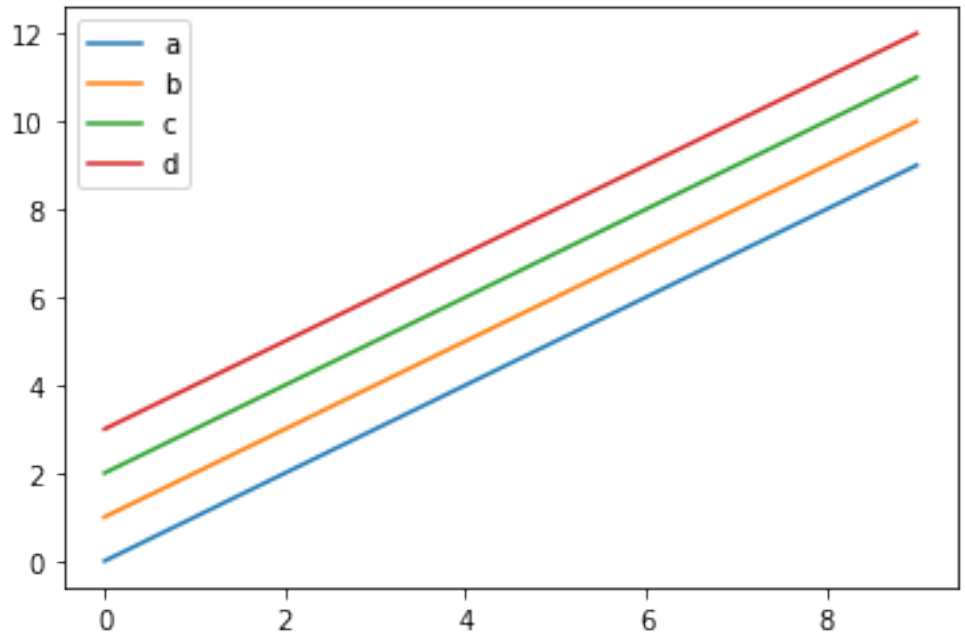
{{df.describe().to_html()}}

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()
```

```
<AxesSubplot:>
```




```
%matplotlib inline
```

The figure above illustrates the information in `df`.

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

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

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

```
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?

```
    __annotations__
```

```
{'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',
 'computational thinking': 'https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2696102/'}
```

```
__annotations__
```

6.1.9 `pidgy` metasyntax

```
    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

```
`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.

````
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.

```alanguage
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.

Non-consecutive header level increase; 0 to 3

Testing code

```
`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

```
`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.

```


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

```
# Front matter
```

```
---
a: foo
---

`import ruamel.yaml` front matter starts with `"---"`.

    assert a == 'foo'
```

`import ruamel.yaml` front matter starts with `"---"`.

```
assert a == 'foo'
```

```
+++
[a]
b="foo"
+++

`import toml` front matter starts with `"+++"`.

    assert a == {'b': 'foo'}
```

`import toml` front matter starts with `"+++"`.

```
assert a == {'b': 'foo'}
```

Emergent language features

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

    markdown_block =\

This is a markdown block.

### 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.

        >>> assert any(x.lstrip().startswith('>>>') for x in my_function.__doc__.
↪splitlines()), "The doctest isn't in the docstring"
```

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

```
markdown_block =\
```

This is a markdown block.

Non-consecutive header level increase; 0 to 3

```
    class MyClass:

The same goes for class definitions.

    ...
    >>> my_function.__doc__
    '`my_function` demonstrates how ...'
```

(continues on next page)

(continued from previous page)

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

```
class MyClass:
```

The same goes for class definitions.

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

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
...
```

6.1.10 `markdown_it` and `pidgy` compatability

```
    from pidgy import util
    import textwrap, markdown_it
```

The `Markdown` class maintains the operational logic to transform partially coded [Markdown] into fully coded Python. The translation happens in two steps:

1. `Markdown.parse`/lex/tokenize the input string to tokens identifying Markdown blocks.
2. `Markdown.render` the tokens into the target language format.

```
    class Markdown(markdown_it.MarkdownIt):
        def parse(self, src, env=None, normalize=False):
            src = enforce_blanklines(src)
            if env is None:
                env = markdown_it.utils.AttrDict()
            env.update(src=src.splitlines(True))
            tokens = super().parse(src, env)
            if normalize: tokens = reconfigure_tokens(filter_tangle_tokens(tokens),␣
↪env)
            return tokens
        def render(self, src, env=None):
            if env is None:
                env  = markdown_it.utils.AttrDict()
            return super().render(src, env)
```

```
    class Renderer:
        __output__ = "html"
        __init__ = markdown_it.renderer.RendererHTML.__init__
```

(continues on next page)

(continued from previous page)

```
        def render(self, tokens, options, env):
            return "".join(env["src"])

        def quote(self, str, trailing=''):
            """Wrap a truple block quotations."""
            quote, length = self.QUOTES[self.QUOTES[0] in str], len(str)
            left, right = length - len(str.lstrip()), len(str.rstrip())
            if not str[left:right].strip(): return str
            if str[right-1] == '\\':
                while str[right-1] == '\\':
                    right -= 1
            else:
                if str[left:right].endswith(quote[0]):
                    quote = {"'''": '"""', '"""': "'''"}[quote]
            return str[:left] + quote + str[left:right] + quote + trailing +␣
↪str[right:]

        def measure_base_indent(self, tokens, env):
            next = self.get_next_code_token(tokens, -1)
            if next and next.type == 'code_block':
                env['base_indent'] = lead_indent(env['src'][slice(*next.map)])
            else:
                env['base_indent'] = 4

        def get_next_code_token(self, tokens, idx):
            for token in tokens[idx+1:]:
                if token.type in {'code_block'}:
                    return token

        def hanging_indent(self, str, env):
            start = len(str)-len(str.lstrip())
            return str[:start] + ' '* env['extra_indent'] + str[start:]

        def indent(self, str, env):
            return textwrap.indent(str, ' ' *env['base_indent'])


        def token_to_str(self, tokens, idx, env):
            if idx < len(tokens):
                if tokens[idx] and tokens[idx].map:
                    return ''.join(env['src'][slice(*tokens[idx].map)])
            return ""

        def update_env(self, code, tokens, idx, env):
            next = self.get_next_code_token(tokens, idx)
            extra_indent = 0
            if next:
                extra_indent = max(0, lead_indent(env['src'][slice(*next.map)]) -env[
↪'base_indent'])
            if not extra_indent and code.rstrip().endswith(":"):
                extra_indent += 4
            rstrip = code.rstrip()
            env.update(
                extra_indent=extra_indent,
                base_indent=util.trailing_indent(code),
                continued=rstrip.endswith('\\'),
                quoted=rstrip.rstrip('\\').endswith(self.QUOTES)
```

(continues on next page)

(continued from previous page)

```
            )
        def render(self, tokens, options, env):
            env.update(base_indent=0, quoted=False, extra_indent=0, continued=False)
            tokens = reconfigure_tokens(filter_tangle_tokens(tokens), env)
            self.measure_base_indent(tokens, env)
            if not tokens:
                return self.quote(''.join(env['src']), trailing=';')
            return textwrap.dedent(continuation(
                markdown_it.renderer.RendererHTML.render(self, tokens, options, env),␣
↪env
            ) + "\n" + self.noncode(tokens, len(tokens), env)).rstrip() + '\n'
```

utility functions for parsing and rendering

```
    import doctest
```

```
    CODE_TYPES = "fence code_block front_matter bullet_list_open ordered_list_open␣
↪footnote_reference_open reference".split()

    def lead_indent(str):
        """Count the lead indent of a string"""
        if not isinstance(str, list):
            str = str.splitlines(True)
        for line in str:
            if line.strip():
                return len(line) - len(line.lstrip())
        return 0



    def filter_tangle_tokens(token, code=None):
        """Filter out tokens that reference a potential coded object."""
        code = code or []
        if isinstance(token, list):
            for token in token:
                code = filter_tangle_tokens(token, code)
        elif token.children:
            for token in token.children:
                code = filter_tangle_tokens(token, code)
        else:
            if token.type in CODE_TYPES:
                if token.type == "code_block":
                    while token.content.lstrip().startswith(">>> "):
                        start, end = next(doctest.DocTestParser._EXAMPLE_RE.
↪finditer(token.content)).span()
                        token.map[0] += len(token.content[:end].splitlines())
                        token.content = token.content[end:]
                    if not token.content.strip(): return code
                if token not in code:
                    code.append(token)
            if code and (code[-1].type == "fence") and code[-1].info:
                code.pop(-1)
        return code or [
            markdown_it.utils.AttrDict(type="code_block", content="", map=(0, 0))
```

(continues on next page)

(continued from previous page)

```
        ]


    def make_reference_tokens(env, *tokens):
        """Turn references in the markdown_it environment to tokens."""
        for reference in env.get("references", {}).values():
            if not tokens:
                tokens += (markdown_it.token.Token("reference", "", 1),)
                tokens[-1].map = reference["map"]
                continue
            for line in env["src"][tokens[-1].map[1] : reference["map"][0]]:
                if line.strip():
                    tokens += (markdown_it.token.Token("reference", "", 1),)
                    tokens[-1].map = reference["map"]
                    break
            else:
                tokens[-1].map[1] = reference["map"][1]

            tokens[-1].content = "".join(env["src"][slice(*tokens[-1].map)])

        return [recontent(x, env) for x in tokens if int.__sub__(*x.map)]


    def recontent(token, env):
        """Update the content on a call."""
        token.content = "".join(env["src"][slice(*token.map)])
        return token


    def reconfigure_tokens(tokens, env):
        """Tokens are miss ordered, this function splits and orders cells."""
        tokens = sorted(tokens + make_reference_tokens(env), key=lambda x: x.map[0])
        new = tokens and [tokens[0]] or []
        for token in tokens[1:]:
            if token.map[0] < new[-1].map[1]:
                new.extend([token, __import__('copy').deepcopy(new[-1])])
                new[-3].map[1], new[-1].map[0] = token.map

                for i in [-3, -1]:
                    (
                        new.pop(i)
                        if int.__sub__(*new[i].map) == 0
                        else recontent(new[i], env)
                    )
                continue
            new.append(token)

        return [x for x in new if int.__sub__(*x.map)]


    def continuation(str, env):
        """Extend a line ending with a continuation."""
        lines, continuing = str.splitlines(), False
        for i, line in enumerate(lines):
            if line.strip():
                continuing = line.endswith("\\")
            elif continuing:
```

(continues on next page)

(continued from previous page)

```
                lines[i] = " " * env["base_indent"] + "\\"
        return "\n".join(lines)

    def enforce_blanklines(str):
        """Make sure blank lines are blank."""
        str = "".join(
            line if line.strip() else "\n" for line in "".join(str).splitlines(True)
        )
        if not str.endswith("\n"):
            str += "\n"
        return str

    def quote_docstrings(str):
        next, end = "", 0
        for m in doctest.DocTestParser._EXAMPLE_RE.finditer(str):
            next += str[slice(end, m.start())] + quote(
                str[slice(m.start(), m.end())], trailing=";"
            )
            end = m.end()
        if next:
            next += str[m.end() :]
        return next or str


    def unfence(str):
        """Remove code fences froma string."""
        return "".join("".join(str.split("```", 1)).rsplit("```", 1))


    def dedent_block(str):
        """Dedent a block of non code."""
        str = textwrap.dedent(str)
        lines = str.splitlines(True)
        for i, line in enumerate(lines):
            if line.strip():
                lines[i] = textwrap.dedent(line)
                break
        return "".join(lines)
```

6.1.11 Test `pidgy.tangle`

```
    import pidgy, ast
```

```
    tangle = pidgy.tangle.Tangle()
```

```
    s = """---
    a: front matter
    ---

    This is a paragraph.

    * a list

        def f():
```

(continues on next page)

(continued from previous page)

```
    A docstring

            print

    """
```

Unnormalized tokens.

```
    tangle.parse(s, normalize=False)
```

```
[Token(type='front_matter', tag='', nesting=0, attrs=None, map=[0, 3], level=0,␣
↪children=None, content='---\na: front matter\n---', markup='---', info='', meta='a:␣
↪front matter', block=True, hidden=True),
 Token(type='paragraph_open', tag='p', nesting=1, attrs=None, map=[4, 5], level=0,␣
↪children=None, content='', markup='', info='', meta={}, block=True, hidden=False),
 Token(type='inline', tag='', nesting=0, attrs=None, map=[4, 5], level=1,␣
↪children=[Token(type='text', tag='', nesting=0, attrs=None, map=None, level=0,␣
↪children=None, content='This is a paragraph.', markup='', info='', meta={},␣
↪block=False, hidden=False)], content='This is a paragraph.', markup='', info='',␣
↪meta={}, block=True, hidden=False),
 Token(type='paragraph_close', tag='p', nesting=-1, attrs=None, map=None, level=0,␣
↪children=None, content='', markup='', info='', meta={}, block=True, hidden=False),
 Token(type='bullet_list_open', tag='ul', nesting=1, attrs=None, map=[6, 10], level=0,
↪ children=None, content='', markup='*', info='', meta={}, block=True, hidden=False),
 Token(type='list_item_open', tag='li', nesting=1, attrs=None, map=[6, 10], level=1,␣
↪children=None, content='', markup='*', info='', meta={}, block=True, hidden=False),
 Token(type='paragraph_open', tag='p', nesting=1, attrs=None, map=[6, 7], level=2,␣
↪children=None, content='', markup='', info='', meta={}, block=True, hidden=False),
 Token(type='inline', tag='', nesting=0, attrs=None, map=[6, 7], level=3,␣
↪children=[Token(type='text', tag='', nesting=0, attrs=None, map=None, level=0,␣
↪children=None, content='a list', markup='', info='', meta={}, block=False,␣
↪hidden=False)], content='a list', markup='', info='', meta={}, block=True,␣
↪hidden=False),
 Token(type='paragraph_close', tag='p', nesting=-1, attrs=None, map=None, level=2,␣
↪children=None, content='', markup='', info='', meta={}, block=True, hidden=False),
 Token(type='paragraph_open', tag='p', nesting=1, attrs=None, map=[8, 9], level=2,␣
↪children=None, content='', markup='', info='', meta={}, block=True, hidden=False),
 Token(type='inline', tag='', nesting=0, attrs=None, map=[8, 9], level=3,␣
↪children=[Token(type='text', tag='', nesting=0, attrs=None, map=None, level=0,␣
↪children=None, content='def f():', markup='', info='', meta={}, block=False,␣
↪hidden=False)], content='def f():', markup='', info='', meta={}, block=True,␣
↪hidden=False),
 Token(type='paragraph_close', tag='p', nesting=-1, attrs=None, map=None, level=2,␣
↪children=None, content='', markup='', info='', meta={}, block=True, hidden=False),
 Token(type='list_item_close', tag='li', nesting=-1, attrs=None, map=None, level=1,␣
↪children=None, content='', markup='*', info='', meta={}, block=True, hidden=False),
 Token(type='bullet_list_close', tag='ul', nesting=-1, attrs=None, map=None, level=0,␣
↪children=None, content='', markup='*', info='', meta={}, block=True, hidden=False),
 Token(type='paragraph_open', tag='p', nesting=1, attrs=None, map=[10, 11], level=0,␣
↪children=None, content='', markup='', info='', meta={}, block=True, hidden=False),
 Token(type='inline', tag='', nesting=0, attrs=None, map=[10, 11], level=1,␣
↪children=[Token(type='text', tag='', nesting=0, attrs=None, map=None, level=0,␣
↪children=None, content='A docstring', markup='', info='', meta={}, block=False,␣
↪hidden=False)], content='A docstring', markup='', info='', meta={}, block=True,␣
↪hidden=False),
```

(continues on next page)

(continued from previous page)

```
 Token(type='paragraph_close', tag='p', nesting=-1, attrs=None, map=None, level=0,␣
↪children=None, content='', markup='', info='', meta={}, block=True, hidden=False),
 Token(type='code_block', tag='code', nesting=0, attrs=None, map=[12, 13], level=0,␣
↪children=None, content='    print\n', markup='', info='', meta={}, block=True,␣
↪hidden=False)]
```

Normalized block tokens

```
    tangle.parse(s, normalize=True)
```

```
[Token(type='front_matter', tag='', nesting=0, attrs=None, map=[0, 3], level=0,␣
↪children=None, content='---\na: front matter\n---', markup='---', info='', meta='a:␣
↪front matter', block=True, hidden=True),
 Token(type='bullet_list_open', tag='ul', nesting=1, attrs=None, map=[6, 10], level=0,
↪ children=None, content='', markup='*', info='', meta={}, block=True, hidden=False),
 Token(type='code_block', tag='code', nesting=0, attrs=None, map=[12, 13], level=0,␣
↪children=None, content='    print\n', markup='', info='', meta={}, block=True,␣
↪hidden=False)]
```

Normalized block tokens

```
    print(tangle.render(s))
```

```
locals().update(__import__('ruamel.yaml').yaml.safe_load("""---
a: front matter
---
""".partition('---')[2].rpartition('---')[0]))
"""* a list

    def f():""";

"""A docstring"""

print
```

```
    transform = pidgy.tangle.pidgyManager().transform_cell
```

```
    print(transform(s))
```

```
locals().update(__import__('ruamel.yaml').yaml.safe_load("""---
a: front matter
---
""".partition('---')[2].rpartition('---')[0]))
"""* a list

    def f():""";

"""A docstring"""

print
```

```
    print(pidgy.tangle.demojize("""
        = 10
    """))
```

```
    _robot_face__panda_face_  = 10
```

```
    ast.parse(transform("""
        return 100
    """)).body
```

```
[<_ast.Return at 0x7f65a425df50>]
```

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

```
<_ast.Call at 0x7f65a4313e50>
```

```
    print(tangle.render("""This is a string \\




        .lower()"""))
```

```
"""This is a string """\
\
\
\
\
.lower()
```

```
    print(tangle.render("""---
    a: 10
    ---

        a=\\

    [a]: xxx
    [b]: ttt


        """))
```

```
locals().update(__import__('ruamel.yaml').yaml.safe_load("""---
a: 10
---
""".partition('---')[2].rpartition('---')[0]))

a=\
\
{x.group(1): x.group(2).rstrip() for x in __import__('pidgy').util.link_item.finditer(
↪"""[a]: xxx
[b]: ttt""")}
```

6.1.12 transclusion of data with jinja2/templates

```
    import pidgy.base, jinja2.meta, IPython, sys, traitlets, functools
```

```
import pidgy.base, jinja2.meta, IPython, sys, traitlets, functools
```

```
    minify = lambda x: __import__('htmlmin').minify(x, False, True, True, True, True,␣
↪True, True)
```

```
minify = lambda x: __import__('htmlmin').minify(x, False, True, True, True, True,␣
↪True, True)
```

```
    def active_types(shell=None):
        shell = shell or IPython.get_ipython()
        if shell:
            object = list(shell.display_formatter.active_types)
            object.insert(object.index('text/html'), object.pop(object.index('text/
↪latex')))
            return reversed(object)
        return []
```

```
def active_types(shell=None):
    shell = shell or IPython.get_ipython()
    if shell:
        object = list(shell.display_formatter.active_types)
        object.insert(object.index('text/html'), object.pop(object.index('text/latex
↪')))
        return reversed(object)
    return []
```

```
    def environment(shell=None):
        environment = __import__('nbconvert').exporters.TemplateExporter().environment
        environment.loader.loaders.append(jinja2.FileSystemLoader('.'))
        environment.display_manager = DisplayManager()
        environment.finalize = Finalize(parent=shell or IPython.get_ipython())
        return environment
```

```
def environment(shell=None):
    environment = __import__('nbconvert').exporters.TemplateExporter().environment
    environment.loader.loaders.append(jinja2.FileSystemLoader('.'))
    environment.display_manager = DisplayManager()
    environment.finalize = Finalize(parent=shell or IPython.get_ipython())
    return environment
```

```
    class MarkdownDisplay(pidgy.base.Display):
        body = traitlets.Unicode()

        vars = traitlets.Set()
        template = traitlets.Any()
        _display = traitlets.Any()
        @traitlets.default('template')
        def _default_template(self):
            return self.environment.from_string(self.body)
        @traitlets.default('vars')
```

(continues on next page)

(continued from previous page)

```
        def _default_vars(self):
            return jinja2.meta.find_undeclared_variables(self.template.environment.
↪parse(self.body))

        def render(self, **kwargs):
            return IPython.display.Markdown(self.template.render(**kwargs))
```

```
class MarkdownDisplay(pidgy.base.Display):
    body = traitlets.Unicode()

    vars = traitlets.Set()
    template = traitlets.Any()
    _display = traitlets.Any()
    @traitlets.default('template')
    def _default_template(self):
        return self.environment.from_string(self.body)
    @traitlets.default('vars')
    def _default_vars(self):
        return jinja2.meta.find_undeclared_variables(self.template.environment.
↪parse(self.body))

    def render(self, **kwargs):
        return IPython.display.Markdown(self.template.render(**kwargs))
```

```
    class DisplayManager(pidgy.base.Trait):
        display = traitlets.Dict()
        state = traitlets.Dict()
        widgets = traitlets.List()
        def pre_execute(self):
                deleted = getattr(self.parent, '_last_parent', {}).get('metadata', {}
↪).get('deletedCells', [])
                for key, displays in self.display.items() if deleted else []:
                    self.display[key] = [
                        x for x in displays if x._display and x._display.display_id␣
↪not in deleted
                    ]

        def append(self, object):
            for key in object.vars:
                self.display[key] = self.display.get(key, [])
                self.display[key].append(object)
                self.state[key] = self.parent.user_ns.get(key, None)

        def pop(self, object):
            for key, values in self.display.items():
                self.display[key] = [x for x in values if x is not object]

        def _post_execute_widget(self, object, change):
            with object.hold_trait_notifications():
                self.post_execute()

        def post_execute(self):
            if not self.enabled: return
            update = {
                x: self.parent.user_ns.get(x, None) for x in self.display
                if isinteractive(self.parent.user_ns.get(x, None)) or
```

(continues on next page)

(continued from previous page)

```
                self.parent.user_ns.get(x, None) is not self.state.get(x, None)
            }
            for key, object in update.items():
                if isinteractive(object) and object not in self.widgets:
                    object.observe(functools.partial(self._post_execute_widget,␣
↪object))
                    self.widgets += [object]
            self.state.update(update)
            for object in set(
                sum([self.display[x] for x in update], [])
            ):
                try:
                    object.update(**self.state)
                except Exception as e:
                    self.pop(object)
                    sys.stderr.writelines(str(self.state))
                    sys.stderr.writelines(str(e).splitlines())
```

```
class DisplayManager(pidgy.base.Trait):
    display = traitlets.Dict()
    state = traitlets.Dict()
    widgets = traitlets.List()
    def pre_execute(self):
            deleted = getattr(self.parent, '_last_parent', {}).get('metadata', {}).
↪get('deletedCells', [])
            for key, displays in self.display.items() if deleted else []:
                self.display[key] = [
                    x for x in displays if x._display and x._display.display_id not␣
↪in deleted
                ]

    def append(self, object):
        for key in object.vars:
            self.display[key] = self.display.get(key, [])
            self.display[key].append(object)
            self.state[key] = self.parent.user_ns.get(key, None)

    def pop(self, object):
        for key, values in self.display.items():
            self.display[key] = [x for x in values if x is not object]

    def _post_execute_widget(self, object, change):
        with object.hold_trait_notifications():
            self.post_execute()

    def post_execute(self):
        if not self.enabled: return
        update = {
            x: self.parent.user_ns.get(x, None) for x in self.display
            if isinteractive(self.parent.user_ns.get(x, None)) or
            self.parent.user_ns.get(x, None) is not self.state.get(x, None)
        }
        for key, object in update.items():
            if isinteractive(object) and object not in self.widgets:
                object.observe(functools.partial(self._post_execute_widget, object))
                self.widgets += [object]
```

(continues on next page)

(continued from previous page)

```
        self.state.update(update)
        for object in set(
            sum([self.display[x] for x in update], [])
        ):
            try:
                object.update(**self.state)
            except Exception as e:
                self.pop(object)
                sys.stderr.writelines(str(self.state))
                sys.stderr.writelines(str(e).splitlines())
```

```
    class Finalize(pidgy.base.Trait):
        def normalize(self, key, object, metadata):
            if key.startswith('image'):
                if 'svg' in key: return minify(object)
                width, height = metadata.get(key, {}).get('width'), metadata.get(key,
↪{}).get('height')
                if isinstance(object, bytes):
                    object = __import__('base64').b64encode(object).decode('utf-8')
                object = F"""<img src="
↪{object}"/>"""
            if key == 'text/html': object = minify(object)
            return object

        def __call__(self, object):
            datum = self.parent.display_formatter.format(object)
            data, metadata = datum if isinstance(datum, tuple) else (datum, {})
            try: key = next(filter(data.__contains__, active_types(self.parent)))
            except StopIteration: return str(object)
            if key == 'text/plain': return str(object)
            return self.normalize(key, data[key], metadata)
```

```
class Finalize(pidgy.base.Trait):
    def normalize(self, key, object, metadata):
        if key.startswith('image'):
            if 'svg' in key: return minify(object)
            width, height = metadata.get(key, {}).get('width'), metadata.get(key, {}).
↪get('height')
            if isinstance(object, bytes):
                object = __import__('base64').b64encode(object).decode('utf-8')
            object = F"""<img src="{object}
↪"/>"""
        if key == 'text/html': object = minify(object)
        return object

    def __call__(self, object):
        datum = self.parent.display_formatter.format(object)
        data, metadata = datum if isinstance(datum, tuple) else (datum, {})
        try: key = next(filter(data.__contains__, active_types(self.parent)))
        except StopIteration: return str(object)
        if key == 'text/plain': return str(object)
        return self.normalize(key, data[key], metadata)
```

```
    def iswidget(object):
        if 'ipywidgets' in sys.modules:
```

(continues on next page)

(continued from previous page)

```
            if isinstance(object, __import__('ipywidgets').Widget):
                return True
        return False

    def ispanel(object):
        if 'param' in sys.modules:
            if isinstance(object, __import__('param').Parameterized):
                return True
        return False

    def isinteractive(object):
        return iswidget(object) | ispanel(object)
```

```
def iswidget(object):
    if 'ipywidgets' in sys.modules:
        if isinstance(object, __import__('ipywidgets').Widget):
            return True
    return False

def ispanel(object):
    if 'param' in sys.modules:
        if isinstance(object, __import__('param').Parameterized):
            return True
    return False

def isinteractive(object):
    return iswidget(object) | ispanel(object)
```

6.1.13 Test `pidgy.weave`

```
    import pidgy, IPython
```

```
import pidgy, IPython
```

```
    weave = pidgy.weave.Weave(parent=IPython.get_ipython())
```

```
weave = pidgy.weave.Weave(parent=IPython.get_ipython())
```

```
    s = """A string to template with a variable: {{foo|default('default')}}."""
```

```
s = """A string to template with a variable: default."""
```

```
    >>> weave.render(s)
    'A string to template with a variable: default.'
```

```
>>> weave.render(s)
'A string to template with a variable: default.'
```

```
    foo = 10
    >>> weave.render(s)
    'A string to template with a variable: 10.'
```

```
foo = 10
>>> weave.render(s)
'A string to template with a variable: 10.'
```

```
    'A string to template with a variable: 900.'
```

```
'A string to template with a variable: 900.'
```

```
'A string to template with a variable: 900.'
```

```
    >>> assert False
    >>> assert 0
```

```
>>> assert False
>>> assert 0
```

```
Traceback (most recent call last):
  File "/home/tonyfast/miniconda3/lib/python3.7/doctest.py", line 2197, in runTest
    raise self.failureException(self.format_failure(new.getvalue()))
AssertionError: Failed doctest test for In[7]
  File "In[7]", line 1, in In[7]

----------------------------------------------------------------------
File "In[7]", line 2, in In[7]
Failed example:
    assert False
Exception raised:
    Traceback (most recent call last):
      File "/home/tonyfast/miniconda3/lib/python3.7/doctest.py", line 1330, in __run
        compileflags, 1), test.globs)
      File "In[9]", line 1, in <module>
    AssertionError
----------------------------------------------------------------------
File "In[7]", line 3, in In[7]
Failed example:
    assert 0
Exception raised:
    Traceback (most recent call last):
      File "/home/tonyfast/miniconda3/lib/python3.7/doctest.py", line 1330, in __run
        compileflags, 1), test.globs)
      File "In[9]", line 1, in <module>
    AssertionError
```

```
Load variables that are defined in the templated_document.md.

    import sys; foo = "default"
    sys.argv = ['empty']

{% include "templated_document.md" %}
```

Load variables that are defined in the templated_document.md.

```
import sys; foo = "default"
sys.argv = ['empty']
```

#!/usr/bin/env python3 -m pidgy template

`foo` is defined as default

My document recieved ['empty'] as arguments.

```
https://en.wikipedia.org/wiki/Bauhaus
```

```
<IPython.lib.display.IFrame at 0x7f524970ba50>
```

6.1.14 test `pidgy`'s interactive testing

```
    import pidgy, unittest, IPython, pytest
    with pidgy.pidgyLoader(lazy=True):
        import pidgy.compat.unittesting
    not_a_test = pytest.mark.skip(reason="This is not a real test")
```

```
import pidgy, unittest, IPython, pytest
with pidgy.pidgyLoader(lazy=True):
    import pidgy.compat.unittesting
not_a_test = pytest.mark.skip(reason="This is not a real test")
```

```
    def test_true():
        assert True
```

```
def test_true():
    assert True
```

```
    @not_a_test
    def test_false():
        assert False
```

```
@not_a_test
def test_false():
    assert False
```

```
Traceback (most recent call last):
  File "<ipython-input-3-ee1396f413a8>", line 3, in test_false
    assert False
AssertionError
```

```
    class tester(unittest.TestCase):
        def test_true(x):
            assert True
        @not_a_test
        def test_false(x):
            assert False
```

```
class tester(unittest.TestCase):
    def test_true(x):
        assert True
    @not_a_test
    def test_false(x):
        assert False
```

```
Traceback (most recent call last):
  File "<ipython-input-4-bf6bf791fc90>", line 6, in test_false
    assert False
AssertionError
```

```
    def test_suite():
        suite = pidgy.testing.Testing(
            parent=IPython.get_ipython()
        ).collect(tester, test_true, test_false, vars=locals(), name=__name__)
        assert len(suite._tests) == 3
        assert suite.run(unittest.TestResult()).failures
```

```
def test_suite():
    suite = pidgy.testing.Testing(
        parent=IPython.get_ipython()
    ).collect(tester, test_true, test_false, vars=locals(), name=__name__)
    assert len(suite._tests) == 3
    assert suite.run(unittest.TestResult()).failures
```