

---

# **Alfred-Workflow Documentation**

***Release 2.0.0-alpha***

**Dean Jackson <[deanishe@deanishe.net](mailto:deanishe@deanishe.net)>**

July 07, 2015



|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Features</b>                                    | <b>3</b>  |
| <b>2</b> | <b>Quick example</b>                               | <b>5</b>  |
| <b>3</b> | <b>Installation</b>                                | <b>7</b>  |
| 3.1      | Installation . . . . .                             | 7         |
| <b>4</b> | <b>The Alfred-Workflow Tutorial</b>                | <b>9</b>  |
| 4.1      | Tutorial . . . . .                                 | 9         |
| <b>5</b> | <b>User Manual</b>                                 | <b>43</b> |
| 5.1      | User Manual . . . . .                              | 43        |
| <b>6</b> | <b>API documentation</b>                           | <b>67</b> |
| 6.1      | Alfred-Workflow API . . . . .                      | 67        |
| <b>7</b> | <b>Script Filter results and the XML format</b>    | <b>81</b> |
| 7.1      | Script Filter Results and the XML Format . . . . . | 81        |
| <b>8</b> | <b>Workflows using Alfred-Workflow</b>             | <b>89</b> |
| 8.1      | Workflows using Alfred-Workflow . . . . .          | 89        |
| <b>9</b> | <b>Feedback, questions, bugs, feature requests</b> | <b>93</b> |
|          | <b>Python Module Index</b>                         | <b>95</b> |



Alfred-Workflow is a Python helper library for [Alfred 2](#) workflow authors, developed and hosted on [GitHub](#).

Alfred workflows typically take user input, fetch data from the Web or elsewhere, filter them and display results to the user. Alfred-Workflow takes care of a lot of the details for you, allowing you to concentrate your efforts on your workflow's functionality.

Alfred-Workflow supports OS X 10.6+ (Python 2.6 and 2.7)



---

## Features

---

- Catches and logs workflow errors for easier development and support
- “*Magic*” *arguments* to help development, debugging and management of the workflow
- *Auto-saves settings*
- Super-simple *data caching*
- Fuzzy, *Alfred-like search/filtering* with *diacritic folding*
- *Keychain support* for secure storage (and syncing) of passwords, API keys etc.
- Simple generation of Alfred feedback (XML output)
- *Input/output decoding* for handling non-ASCII text
- *Lightweight web* API with *Requests*-like interface
- Pre-configured logging
- Painlessly add directories to `sys.path`
- Easily launch *background tasks* (daemons) to keep your workflow responsive
- *Check for and install new workflow versions* using GitHub releases.





---

## Quick example

---

Here's how to show recent [Pinboard.in](#) posts in Alfred.

Create a new workflow in Alfred's preferences. Add a **Script Filter** with Language `/usr/bin/python` and paste the following into the **Script** field (changing `API_KEY`):

```
1 import sys
2 from workflow import Workflow, ICON_WEB, web
3
4 API_KEY = 'your-pinboard-api-key'
5
6 def main(wf):
7     url = 'https://api.pinboard.in/v1/posts/recent'
8     params = dict(auth_token=API_KEY, count=20, format='json')
9     r = web.get(url, params)
10    r.raise_for_status()
11    for post in r.json()['posts']:
12        wf.add_item(post['description'], post['href'], arg=post['href'],
13                    uid=post['hash'], valid=True, icon=ICON_WEB)
14    wf.send_feedback()
15
16
17 if __name__ == u"__main__":
18     wf = Workflow()
19     sys.exit(wf.run(main))
```

Add an **Open URL** action to your workflow with `{query}` as the **URL**, connect your **Script Filter** to it, and you can now hit **ENTER** on a Pinboard item in Alfred to open it in your browser.

**Warning:** Using the above example code as a workflow will likely get you banned by the Pinboard API. See the [Tutorial](#) if you want to build an API terms-compliant (and super-fast) Pinboard workflow.



---

## Installation

---

Alfred-Workflow can be installed from the [Python Package Index](#) with `pip` or from the source on [GitHub](#).

### 3.1 Installation

Alfred-Workflow can be installed from the [Python Package Index](#) with `pip` or from the source code on [GitHub](#).

#### 3.1.1 `pip` / PyPi

You can install Alfred-Workflow directly into your workflow with:

```
pip install --target=/path/to/my/workflow Alfred-Workflow
```

**Important:** If you intend to distribute your workflow to other users, you should include Alfred-Workflow (and other non-standard Python libraries your workflow requires) within your workflow as described above. **Do not** ask users to install anything into their system Python. That way lies broken software.

#### 3.1.2 GitHub

Download the `alfred-workflow-X.X.X.zip` file from the [GitHub releases](#) page and either extract the ZIP to the root directory of your workflow (where `info.plist` is) or place the ZIP in the root directory and add `sys.path.insert(0, 'alfred-workflow-X.X.X.zip')` to the top of your Python scripts.

**Important:** `background` and `update` will not work if you are importing Alfred-Workflow from a zip file.

If you need to use `background` or the self-updating functionality, you *must* extract the zip archive.

Alternatively, you can download the [source code](#) from the [GitHub repository](#) and copy the `workflow` subfolder to the root directory of your workflow.

Your Workflow directory should look something like this (where `yourscript.py` contains your workflow code and `info.plist` is the workflow information file generated by Alfred):

```
Your Workflow/  
  info.plist  
  icon.png  
  workflow/  
    __init__.py  
    background.py  
    update.py  
    version  
    workflow.py
```

```
web.py
yourscript.py
etc.
```

Or like this:

```
Your Workflow/
  info.plist
  icon.png
  workflow-1.X.X.zip
  yourscript.py
  etc.
```

---

## The Alfred-Workflow Tutorial

---

A *two-part tutorial* on writing an Alfred workflow with Alfred-Workflow, taking you through the basics to a performant and release-ready workflow. This is the best starting point for workflow authors new to Python or programming in general. More experienced Python coders should skim this or skip straight ahead to the *User Manual*.

### 4.1 Tutorial

This is a two-part tutorial on writing an Alfred 2 workflow with Alfred-Workflow, taking you through the basics to a full-featured workflow ready to share with the world.

#### 4.1.1 Part 1: A Basic Pinboard Workflow

In which we build an Alfred workflow to view recent posts to [Pinboard](#).

If you're new to Alfred and/or coding in general, start here.

##### Part 1: A Basic Pinboard Workflow

In which we build an Alfred workflow to view recent posts to [Pinboard.in](#).

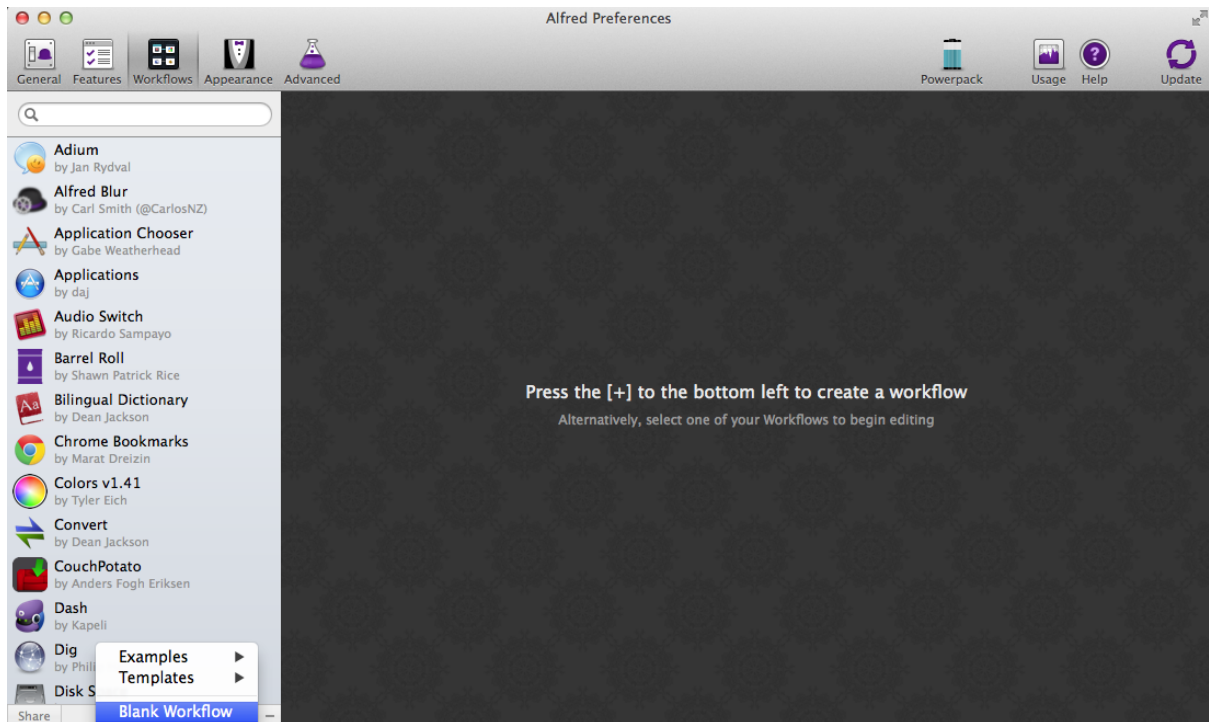
---

**Note:** To use workflows, you must own Alfred's [Powerpack](#).

---

##### Creating a new Workflow

First, create a new, blank workflow in Alfred 2's Preferences, under the **Workflows** tab:

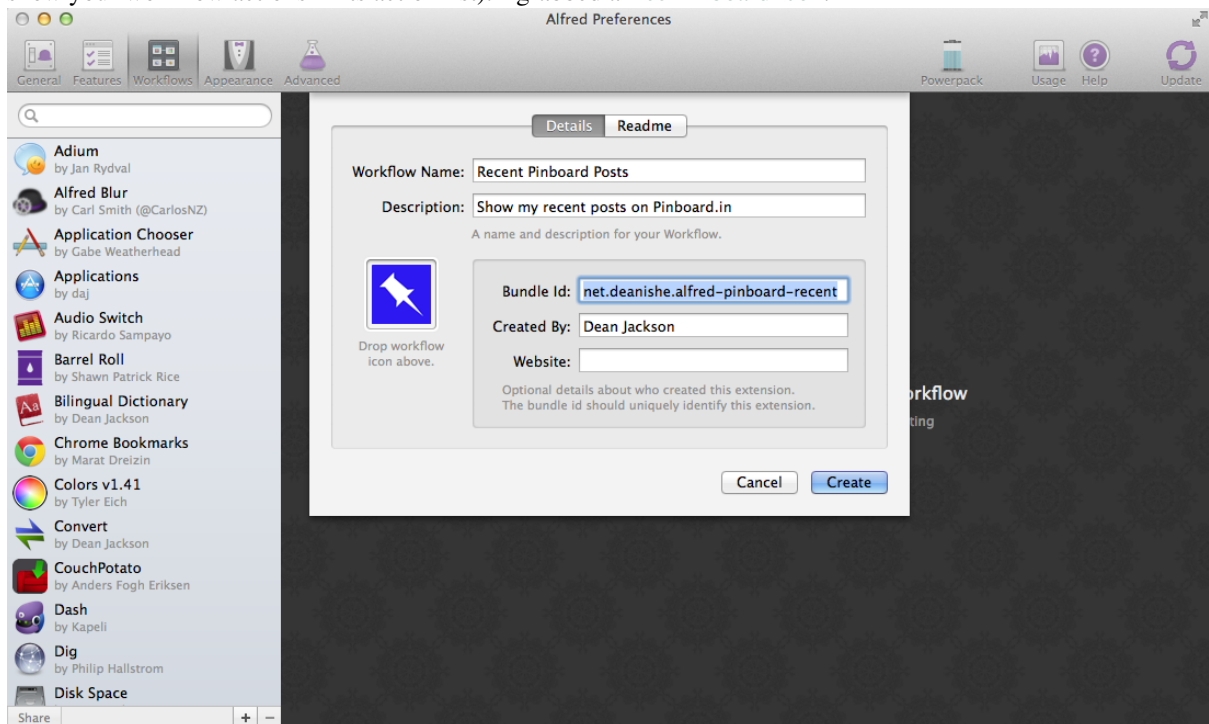


### Describing your Workflow

When the info dialog pops up, give your workflow a name, a **Bundle Id**, and possibly a description.

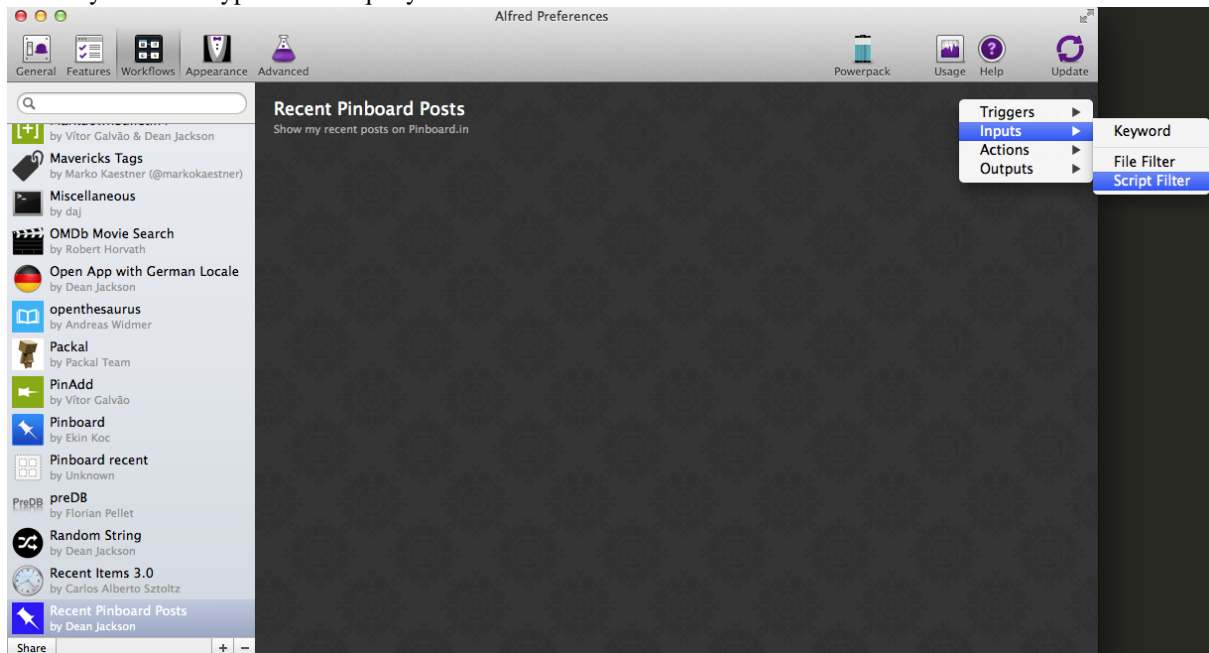
**Important:** The **Bundle Id** is essential: it's the unique name used by Alfred and Alfred-Workflow internally to identify your workflow. Alfred-Workflow won't work without it.

You can also drag an image to the icon field to the left to make your workflow pretty (Alfred will use this icon to show your workflow actions in its action list). I grabbed a [free Pinboard icon](#).

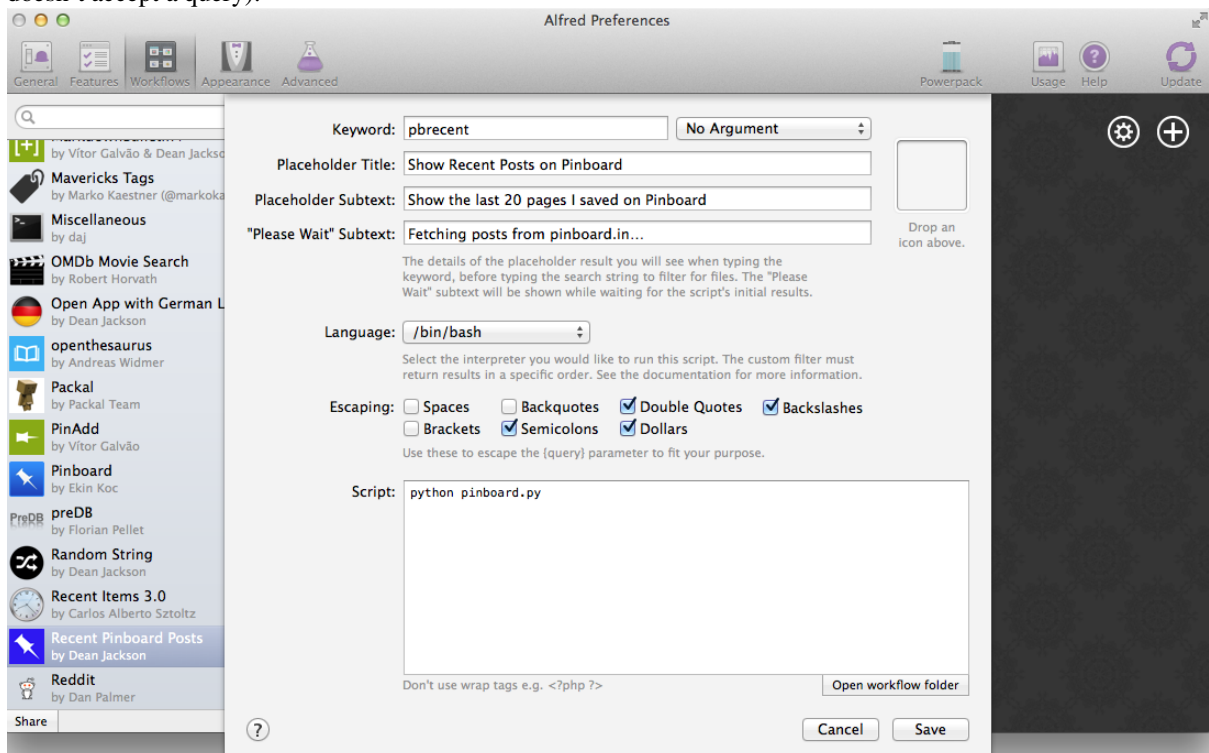


## Adding a Script Filter

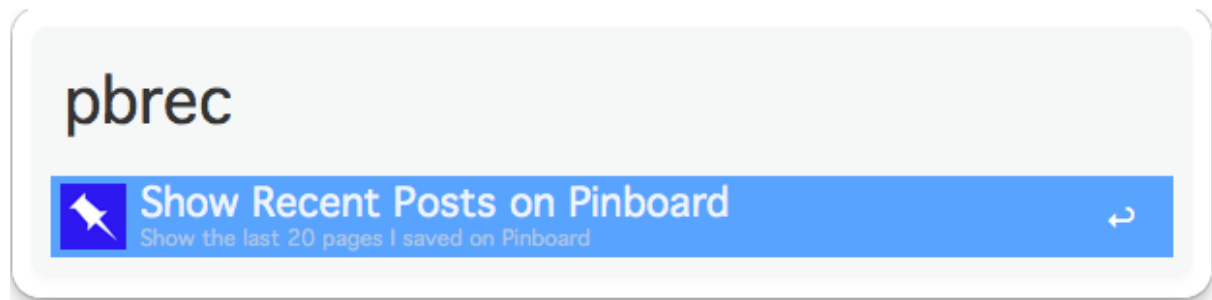
The next step is to add a **Script Filter**. Script Filters receive input from Alfred (the query entered by the user) and send results back to Alfred. They should run as quickly as possible because Alfred will try to call the Script Filter for every character typed into its query box:



And enter the details for this action (the **Escaping** options don't matter at the moment because our script currently doesn't accept a query):



Choose a **Keyword**, which you will enter in Alfred to activate your workflow. At the moment, our Script Filter won't take any arguments, so choose **No Argument**. The **Placeholder Title** and **Subtext** are what Alfred will show when you type the **Keyword**:



The “**Please Wait**” **Subtext** is what is shown when your workflow is working, which in our case means fetching data from pinboard.in.

Very importantly, set the **Language** to `/bin/bash`. The **Script** field should contain:

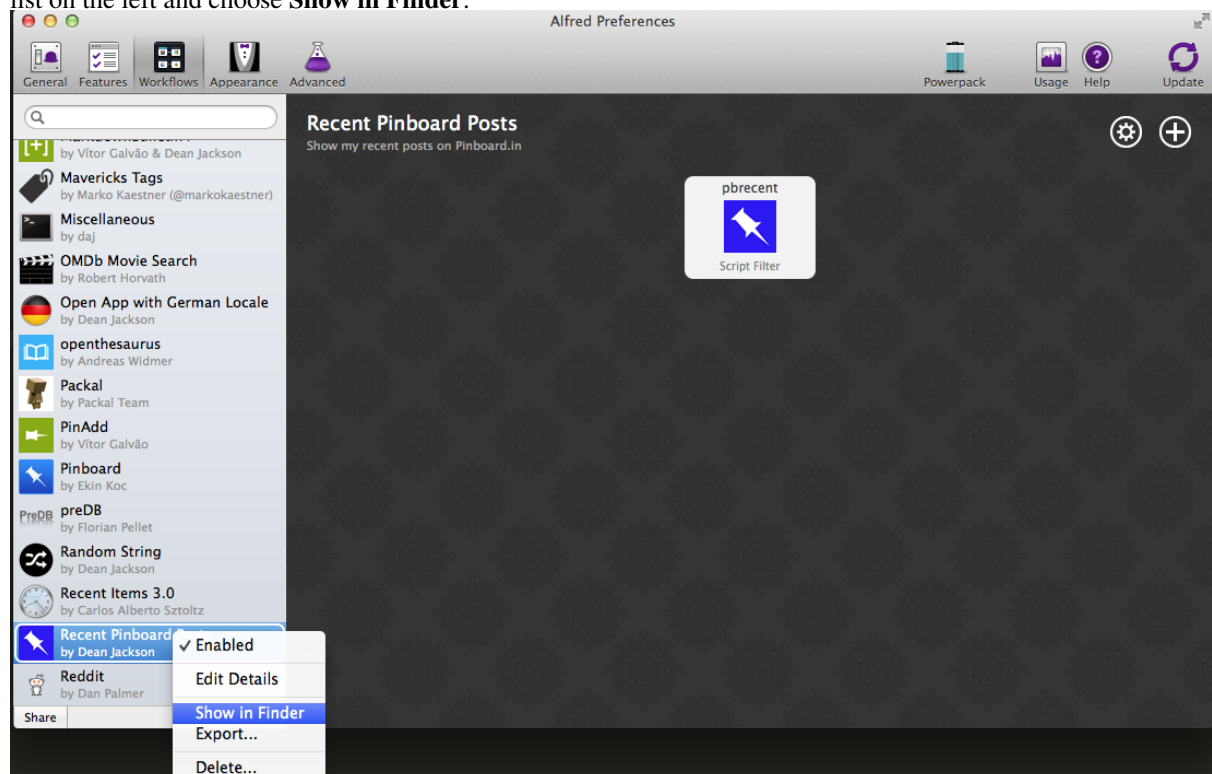
```
python pinboard.py
```

We’re going to create the `pinboard.py` script in a second. The **Escaping** options don’t matter for now because our Script Filter doesn’t accept an argument.

**Note:** You *can* choose `/usr/bin/python` as the **Language** and paste your Python code into the **Script** box, but this isn’t the best idea.

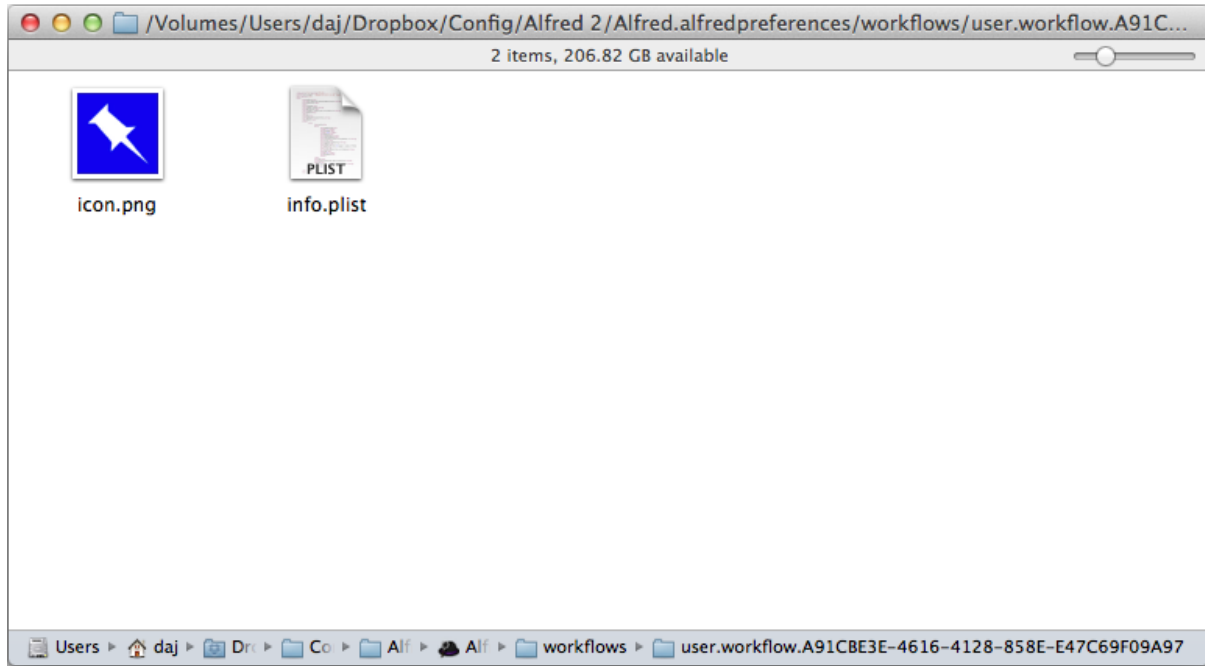
If you do this, you can’t run the script from the Terminal (which can be helpful when developing/debugging), and you can’t as easily use a proper code editor, which makes debugging difficult: Python always tells you which line an error occurred on, but the **Script** field doesn’t show line numbers, so lots of counting is involved.

Now Alfred has created the workflow, we can open it up and add our script. Right-click on your workflow in the list on the left and choose **Show in Finder**.

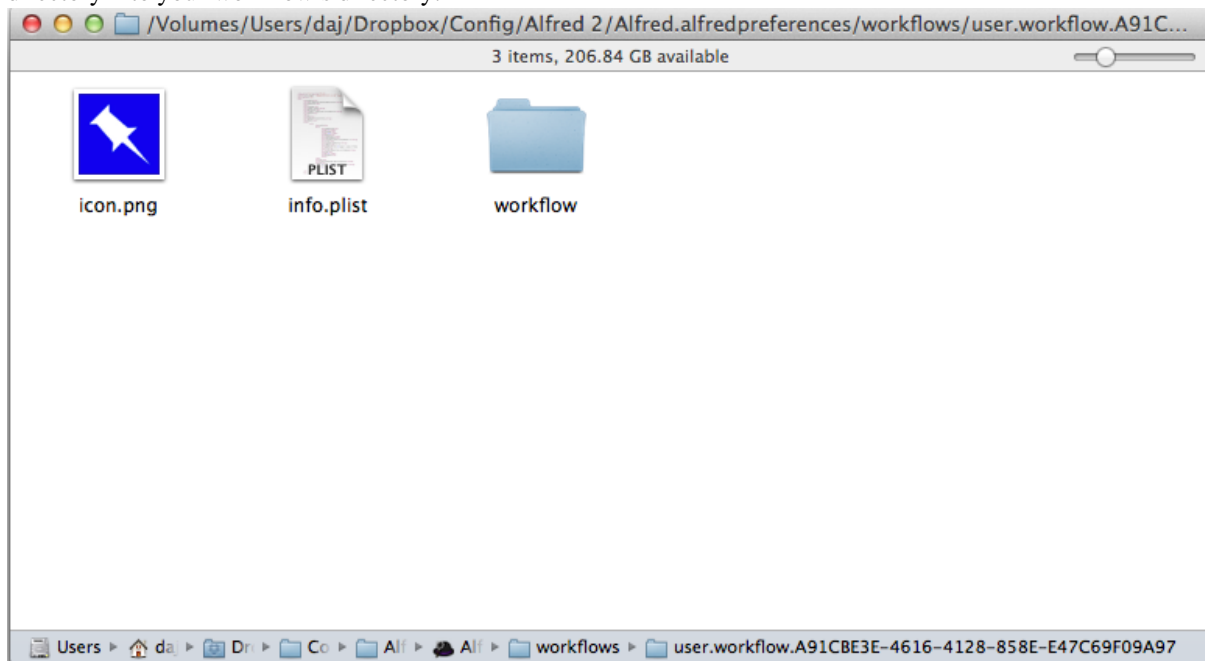


The directory will show one or two files (depending on whether or not you chose an icon):





At this point, download the [latest release of Alfred-Workflow](#) from GitHub, extract it and copy the workflow directory into your workflow's directory:



Now we can start coding.

### Writing your Python script

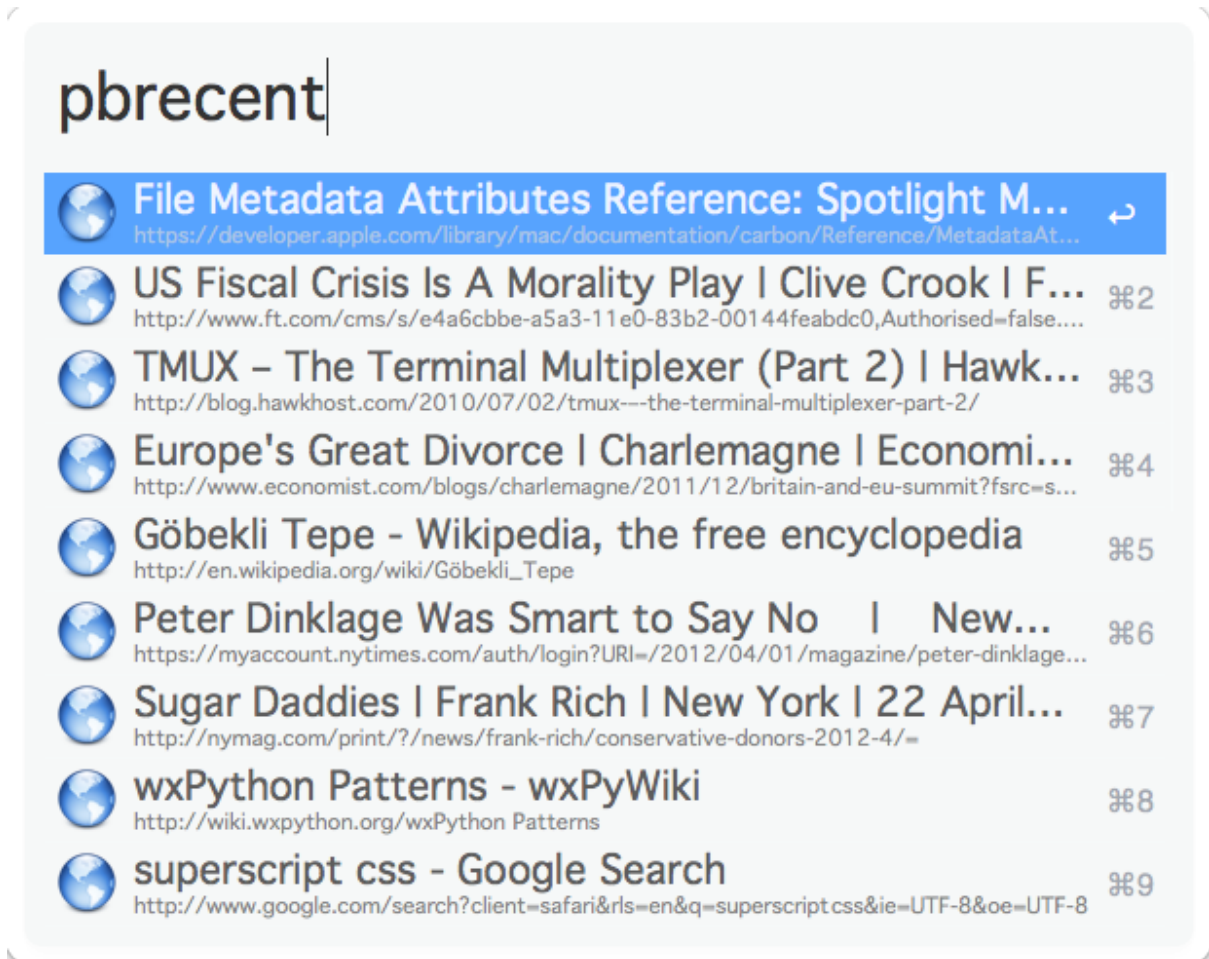
Using your text editor of choice, create a new text file and save it in your workflow directory as `pinboard.py` (the name we used when setting up the Script Filter).

Add the following code to `pinboard.py` (be sure to change `API_KEY` to your pinboard API key. You can find it on the [settings/password](#) page):

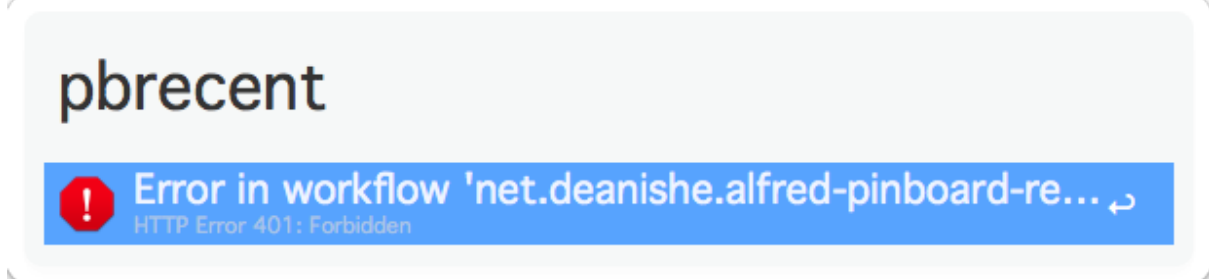
```
1 # encoding: utf-8
2
3 import sys
```

```
4 from workflow import Workflow, ICON_WEB, web
5
6 API_KEY = 'your-pinboard-api-key'
7
8
9 def main(wf):
10     url = 'https://api.pinboard.in/v1/posts/recent'
11     params = dict(auth_token=API_KEY, count=20, format='json')
12     r = web.get(url, params)
13
14     # throw an error if request failed
15     # Workflow will catch this and show it to the user
16     r.raise_for_status()
17
18     # Parse the JSON returned by pinboard and extract the posts
19     result = r.json()
20     posts = result['posts']
21
22     # Loop through the returned posts and add an item for each to
23     # the list of results for Alfred
24     for post in posts:
25         wf.add_item(title=post['description'],
26                     subtitle=post['href'],
27                     icon=ICON_WEB)
28
29     # Send the results to Alfred as XML
30     wf.send_feedback()
31
32
33 if __name__ == u"__main__":
34     wf = Workflow()
35     sys.exit(wf.run(main))
```

All being well, our workflow should now work. Fire up Alfred, enter your keyword and hit **ENTER**. You should see something like this:



If something went wrong (e.g. an incorrect API key, as in the screenshot), you should see an error like this:



If Alfred shows nothing at all, it probably couldn't run your Python script at all. You'll have to [open the workflow directory in Terminal](#) and run the script by hand to see the error:

```
python pinboard.py
```

### Adding workflow actions

So now we can see a list of recent posts in Alfred, but can't do anything with them. We're going to change that and make the items "actionable" (i.e. you can hit **ENTER** on them and something happens, in this case, the page will be opened in your browser).

Add the highlighted lines (27–28) to your `pinboard.py` file:

```
1 # encoding: utf-8
2
3 import sys
4 from workflow import Workflow, ICON_WEB, web
```

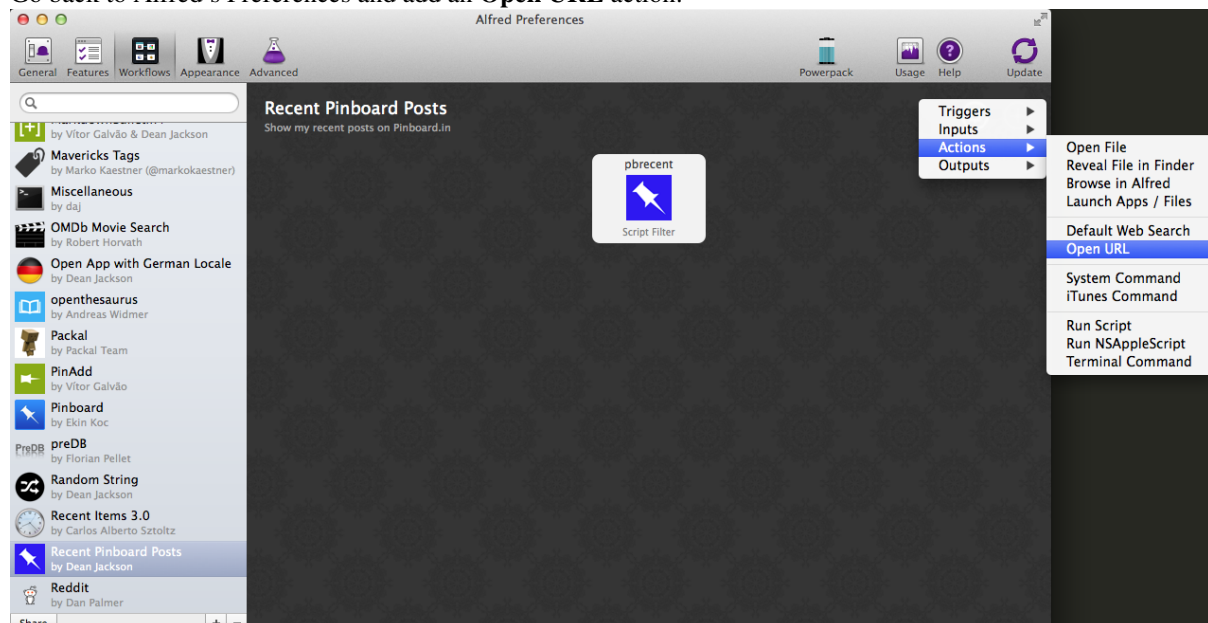
```

5
6 API_KEY = 'your-pinboard-api-key'
7
8
9 def main(wf):
10     url = 'https://api.pinboard.in/v1/posts/recent'
11     params = dict(auth_token=API_KEY, count=20, format='json')
12     r = web.get(url, params)
13
14     # throw an error if request failed
15     # Workflow will catch this and show it to the user
16     r.raise_for_status()
17
18     # Parse the JSON returned by pinboard and extract the posts
19     result = r.json()
20     posts = result['posts']
21
22     # Loop through the returned posts and add an item for each to
23     # the list of results for Alfred
24     for post in posts:
25         wf.add_item(title=post['description'],
26                     subtitle=post['href'],
27                     arg=post['href'],
28                     valid=True,
29                     icon=ICON_WEB)
30
31     # Send the results to Alfred as XML
32     wf.send_feedback()
33
34
35 if __name__ == u"__main__":
36     wf = Workflow()
37     sys.exit(wf.run(main))

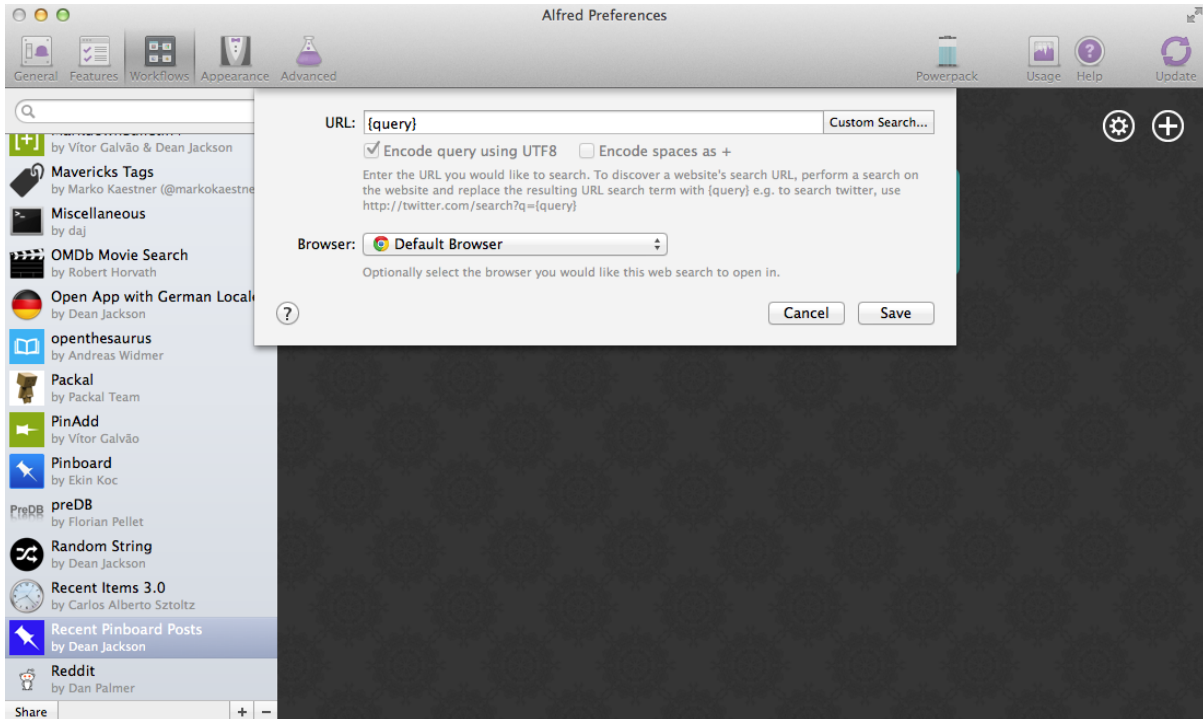
```

valid=True tells Alfred that the item is actionable and arg is the value it will pass to the next action (in this case a URL).

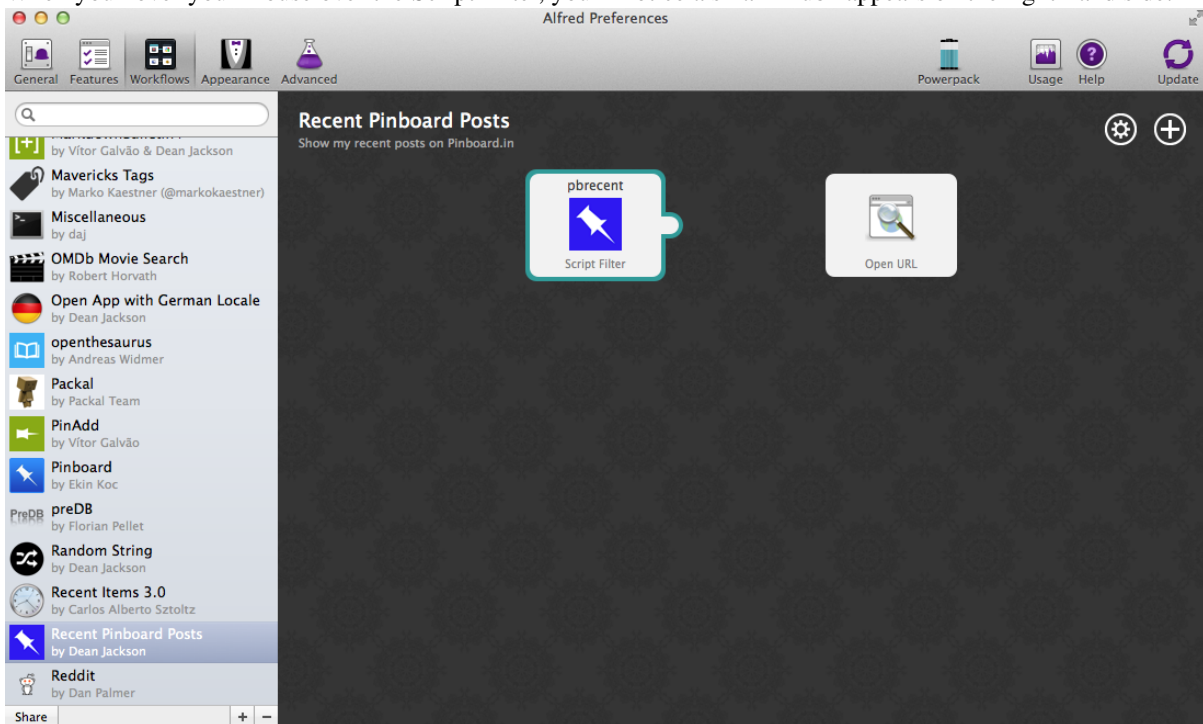
Go back to Alfred's Preferences and add an **Open URL** action:



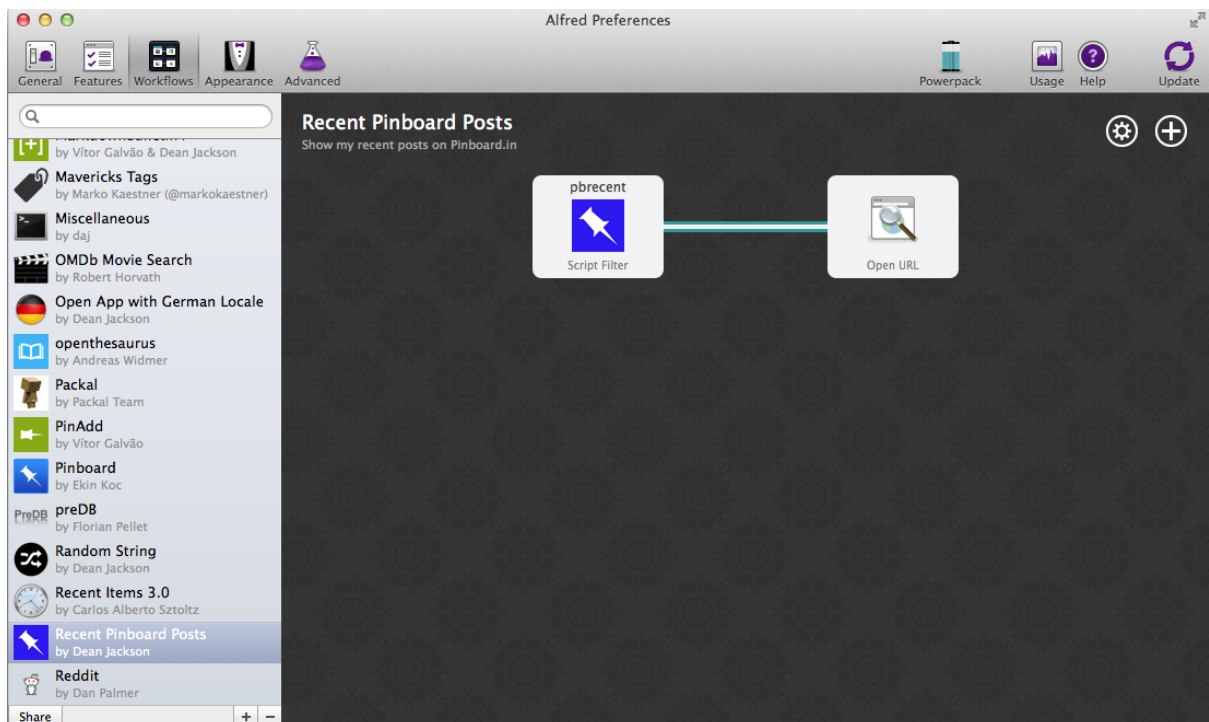
Then enter {query} as the URL:



When you hover your mouse over the Script Filter, you'll notice a small “nub” appears on the right-hand side:



Click and hold on this, and drag a connection to the **Open URL** action:



Now run your workflow again in Alfred, select one of the results and hit **ENTER**. The post's webpage should open in your default browser.

### Improving performance and not getting banned

The terms of use of the Pinboard API specifically limit calls to the recent posts method to **1 call/minute**. As it's likely you'll call your workflow more often than that, we need to cache the results from the API and use the cached data for at least a minute. Alfred-Workflow makes this a doddle with its `cached_data()` method.

Go back to `pinboard.py` and make the following changes:

```

1  # encoding: utf-8
2
3  import sys
4  from workflow import Workflow, ICON_WEB, web
5
6  API_KEY = 'your-pinboard-api-key'
7
8
9  def get_recent_posts():
10     """Retrieve recent posts from Pinboard.in
11
12     Returns a list of post dictionaries.
13
14     """
15     url = 'https://api.pinboard.in/v1/posts/recent'
16     params = dict(auth_token=API_KEY, count=20, format='json')
17     r = web.get(url, params)
18
19     # throw an error if request failed
20     # Workflow will catch this and show it to the user
21     r.raise_for_status()
22
23     # Parse the JSON returned by pinboard and extract the posts
24     result = r.json()
25     posts = result['posts']
26     return posts

```

```

27
28
29 def main(wf):
30
31     # Retrieve posts from cache if available and no more than 60
32     # seconds old
33     posts = wf.cached_data('posts', get_recent_posts, max_age=60)
34
35     # Loop through the returned posts and add an item for each to
36     # the list of results for Alfred
37     for post in posts:
38         wf.add_item(title=post['description'],
39                     subtitle=post['href'],
40                     arg=post['href'],
41                     valid=True,
42                     icon=ICON_WEB)
43
44     # Send the results to Alfred as XML
45     wf.send_feedback()

```

We’ve moved the code that retrieves the data from the API to a separate function (`get_recent_posts()`, line 9) and instead we ask `Workflow.cached_data()` (line 33) for the data cached under the name `posts` (the first argument). `cached_data()` will first check its cache for data saved under `posts` and return those data if they’re less than `max_age` seconds old. If the data are older or don’t exist, it will call the `get_recent_posts()` function passed as the second parameter, cache the data returned by that function under the name `posts` and return it.

So now we won’t get banned by Pinboard for hammering the API, and as a bonus, the workflow is now *blazingly* fast when the data are in its cache. For this reason, it’s probably a good idea to increase `max_age` to 300 or 600 seconds (5 or 10 minutes) or even more—depending on how often you add new posts to Pinboard—to get super-fast results more often.

### Making the posts searchable

What if you’re looking for a specific post? Who’s got time to scroll through a list of 20 results? Let’s make them searchable.

First, update the Script Filter settings. Next to **Keyword**, change **No Argument** to **Argument Optional** and select **with space**. **with space** means that when you hit **ENTER** or **TAB** on your workflow action, Alfred will add a space after it, so you can start typing your query immediately. Then add "{query}" in the **Script** text field. {query} will be replaced by Alfred with whatever you’ve typed after the keyword. Finally, set the **Escaping** options to:

- Backquotes
- Double Quotes
- Dollars
- Backslashes

and **nothing** else. This ensures that the query reaches your Python script unmolested by `bash`. Your **Script Filter** settings should now look like this:



Alfred Preferences

Appearance Advanced Powerpack

Keyword:  ☒ with space

Placeholder Title:

Placeholder Subtext:

"Please Wait" Subtext:

The details of the placeholder result you will see when typing the keyword, before typing the search string to filter for files. The "Please Wait" subtext will be shown while waiting for the script's initial results.

Language:

Select the interpreter you would like to run this script. The custom filter must return results in a specific order. See the documentation for more information.

Escaping: ☐ Spaces ☒ Backquotes ☒ Double Quotes ☒ Backslashes  
☐ Brackets ☐ Semicolons ☒ Dollars

Use these to escape the {query} parameter to fit your purpose.

Script:

Don't use wrap tags e.g. <?php ?>

First, we'll set the script to get 100 recent posts from Pinboard (the maximum allowed) in line 16 and to cache them for 10 minutes in line 33 (or use 300 seconds for 5 minutes if you're a heavy Pinboardista):

```

1  # encoding: utf-8
2
3  import sys
4  from workflow import Workflow, ICON_WEB, web
5
6  API_KEY = 'your-pinboard-api-key'
7
8
9  def get_recent_posts():
10     """Retrieve recent posts from Pinboard.in
11
12     Returns a list of post dictionaries.
13
14     """
15     url = 'https://api.pinboard.in/v1/posts/recent'
16     params = dict(auth_token=API_KEY, count=100, format='json')
17     r = web.get(url, params)
18
19     # throw an error if request failed

```



```

20     # Workflow will catch this and show it to the user
21     r.raise_for_status()
22
23     # Parse the JSON returned by pinboard and extract the posts
24     result = r.json()
25     posts = result['posts']
26     return posts
27
28
29 def main(wf):
30
31     # Retrieve posts from cache if available and no more than 600
32     # seconds old
33     posts = wf.cached_data('posts', get_recent_posts, max_age=600)
34
35     # Loop through the returned posts and add an item for each to
36     # the list of results for Alfred
37     for post in posts:
38         wf.add_item(title=post['description'],
39                     subtitle=post['href'],
40                     arg=post['href'],
41                     valid=True,
42                     icon=ICON_WEB)
43
44     # Send the results to Alfred as XML
45     wf.send_feedback()
46
47
48 if __name__ == u"__main__":
49     wf = Workflow()
50     sys.exit(wf.run(main))

```

Then we need to add the ability to receive the query from Alfred and filter our posts based on it:

```

1  # encoding: utf-8
2
3  import sys
4  from workflow import Workflow, ICON_WEB, web
5
6  API_KEY = 'your-pinboard-api-key'
7
8
9  def get_recent_posts():
10     """Retrieve recent posts from Pinboard.in
11
12     Returns a list of post dictionaries.
13
14     """
15     url = 'https://api.pinboard.in/v1/posts/recent'
16     params = dict(auth_token=API_KEY, count=100, format='json')
17     r = web.get(url, params)
18
19     # throw an error if request failed
20     # Workflow will catch this and show it to the user
21     r.raise_for_status()
22
23     # Parse the JSON returned by pinboard and extract the posts
24     result = r.json()
25     posts = result['posts']
26     return posts
27
28
29 def search_key_for_post(post):

```

```
30     """Generate a string search key for a post"""
31     elements = []
32     elements.append(post['description']) # title of post
33     elements.append(post['tags']) # post tags
34     elements.append(post['extended']) # description
35     return u' '.join(elements)
36
37
38 def main(wf):
39
40     # Get query from Alfred
41     if len(wf.args):
42         query = wf.args[0]
43     else:
44         query = None
45
46     # Retrieve posts from cache if available and no more than 600
47     # seconds old
48     posts = wf.cached_data('posts', get_recent_posts, max_age=600)
49
50     # If script was passed a query, use it to filter posts
51     if query:
52         posts = wf.filter(query, posts, key=search_key_for_post)
53
54     # Loop through the returned posts and add an item for each to
55     # the list of results for Alfred
56     for post in posts:
57         wf.add_item(title=post['description'],
58                     subtitle=post['href'],
59                     arg=post['href'],
60                     valid=True,
61                     icon=ICON_WEB)
62
63     # Send the results to Alfred as XML
64     wf.send_feedback()
65
66
67 if __name__ == u"__main__":
68     wf = Workflow()
69     sys.exit(wf.run(main))
```

Looking at `main()` first, we add a query variable (lines 40–44). Because our Script Filter can run with or without an argument, we test to see if any were passed to the script using via `args` attribute of `Workflow`, and grab the first one if there were (this will be the contents of `{query}` from the Script Filter).

Using `args` is similar to accessing `sys.argv[1:]` directly, but additionally decodes the arguments to Unicode and normalizes them. It also enables “*Magic*” arguments.

After getting all the posts from the cache or Pinboard, we then filter them using the `Workflow.filter()` method if there is a query (lines 51–52).

`Workflow.filter()` implements an Alfred-like search algorithm (e.g. “am” will match “Activity Monitor” as well as “I Am Legend”), but it needs a string to search. Therefore, we write the `search_key_for_post()` (line 29) function that will build a searchable string for each post, comprising its title, tags and description (in that order).

---

**Important:** In the last line of `search_key_for_post()`, we join the elements with `u' '` (a Unicode space), not `' '` (a byte-string space). The `web.Response.json()` method returns Unicode (as do most Alfred-Workflow methods and functions), and mixing Unicode and byte-strings will cause a fatal error if the byte-string contains non-ASCII characters. In this particular situation, using a byte-string space wouldn’t cause any problems (a space is ASCII), but avoiding mixing byte-strings and Unicode is a very good habit to get into.

When coding in Python 2, you have to be aware of which strings are Unicode and which are encoded (byte) strings.

Best practice is to use Unicode internally and decode all text to Unicode when it arrives in your workflow (from the Web, system etc.).

Alfred-Workflow's APIs use Unicode and it works hard to hide as much of the complexity of working with byte-strings and Unicode as possible, but you still need to manually decode encoded byte-strings from other sources with `Workflow.decode()` to avoid fatal encoding errors.

See *Encoded strings and Unicode* in the *User Manual* for more information on dealing with encoded (byte) strings and Unicode in workflows.

---

**Improving the search results** If you've been trying out the workflow, you've probably noticed that your queries match a lot of posts they really shouldn't. The reason for this is that, by default, `Workflow.filter()` matches *anything* that contains all the characters of `query` in the same order, regardless of case. To fix this, we'll add a `min_score` argument to `Workflow.filter()`. Change the line:

```
posts = wf.filter(query, posts, key=search_key_for_post)
```

to:

```
posts = wf.filter(query, posts, key=search_key_for_post, min_score=20)
```

and try the workflow again. The junk results should be gone. You can adjust `min_score` up or down depending on how strict you want to be with the results.

### What now?

So we've got a working workflow, but it's not yet ready to be distributed to other users (we can't reasonably ask users to edit the code to enter their API key, especially as they'd have to do it again after updating the workflow to a new version). We'll turn what we've got into a distribution-ready workflow in the *second part of the tutorial*.

**Further reading** For more information about writing Alfred workflows, try the following:

- [A good tutorial on Alfred workflows for beginners](#) by Richard Guay
- [The Alfred Forum](#). It's a good place to find workflows and the [Workflow Help & Questions](#) forum is the best place to get help with writing workflows.

To learn more about coding in Python, try these resources:

- [The Python Tutorial](#) is a good place to start learning (more) about Python programming.
- [Dive into Python](#) by the dearly departed (from the Web) Mark Pilgrim is a wonderful (and free) book.
- [Learn Python the Hard Way](#) isn't as hard as it sounds. It's actually rather excellent, in fact.

## 4.1.2 Part 2: A Distribution-Ready Pinboard Workflow

In which we make our [Pinboard](#) workflow ready for the masses.

Demonstrates more advanced usage of Alfred-Workflow and a few workflow tricks that might also be of interest to intermediate Pythonistas.

### Part 2: A Distribution-Ready Pinboard Workflow

In which we create a [Pinboard.in](#) workflow ready for mass consumption.

In the *first part* of the tutorial, we built a useable workflow to view, search and open your recent Pinboard posts. The workflow isn't quite ready to be distributed to other users, however: we can't expect them to go grubbing around in the source code like an animal to set their own API keys.

What's more, an update to the workflow would overwrite their changes.

So now we're going to edit the workflow so users can add their API key from the comfort of Alfred's friendly query box and use `Workflow.settings` to save it in the workflow's data directory where it won't get overwritten.

### Performing multiple actions from one script

To set the user's API key, we're going to need a new action. We could write a second script to do this, but we're going to stick with one script and make it smart enough to do two things, instead. The advantage of using one script is that if you build a workflow with lots of actions, you don't have a dozen or more scripts to manage.

We'll start by adding an argument parser (using `argparse`) to `main()` and some if-clauses to alter the script's behaviour depending on the arguments passed to it by Alfred.

```
1  # encoding: utf-8
2
3  import sys
4  import argparse
5  from workflow import Workflow, ICON_WEB, ICON_WARNING, web
6
7
8  def get_recent_posts(api_key):
9      """Retrieve recent posts from Pinboard.in
10
11     Returns a list of post dictionaries.
12
13     """
14     url = 'https://api.pinboard.in/v1/posts/recent'
15     params = dict(auth_token=api_key, count=100, format='json')
16     r = web.get(url, params)
17
18     # throw an error if request failed
19     # Workflow will catch this and show it to the user
20     r.raise_for_status()
21
22     # Parse the JSON returned by pinboard and extract the posts
23     result = r.json()
24     posts = result['posts']
25     return posts
26
27
28  def search_key_for_post(post):
29      """Generate a string search key for a post"""
30     elements = []
31     elements.append(post['description']) # title of post
32     elements.append(post['tags']) # post tags
33     elements.append(post['extended']) # description
34     return u' '.join(elements)
35
36
37  def main(wf):
38
39     # build argument parser to parse script args and collect their
40     # values
41     parser = argparse.ArgumentParser()
42     # add an optional (nargs='?') --setkey argument and save its
43     # value to 'apikey' (dest). This will be called from a separate "Run Script"
44     # action with the API key
45     parser.add_argument('--setkey', dest='apikey', nargs='?', default=None)
46     # add an optional query and save it to 'query'
47     parser.add_argument('query', nargs='?', default=None)
48     # parse the script's arguments
```

```

49 args = parser.parse_args(wf.args)
50
51 #####
52 # Save the provided API key
53 #####
54
55 # decide what to do based on arguments
56 if args.apikey: # Script was passed an API key
57     # save the key
58     wf.settings['api_key'] = args.apikey
59     return 0 # 0 means script exited cleanly
60
61 #####
62 # Check that we have an API key saved
63 #####
64
65 api_key = wf.settings.get('api_key', None)
66 if not api_key: # API key has not yet been set
67     wf.add_item('No API key set.',
68                 'Please use pbsetkey to set your Pinboard API key.',
69                 valid=False,
70                 icon=ICON_WARNING)
71     wf.send_feedback()
72     return 0
73
74 #####
75 # View/filter Pinboard posts
76 #####
77
78 query = args.query
79 # Retrieve posts from cache if available and no more than 600
80 # seconds old
81
82 def wrapper():
83     """`cached_data` can only take a bare callable (no args),
84     so we need to wrap callables needing arguments in a function
85     that needs none.
86     """
87     return get_recent_posts(api_key)
88
89 posts = wf.cached_data('posts', wrapper, max_age=600)
90
91 # If script was passed a query, use it to filter posts
92 if query:
93     posts = wf.filter(query, posts, key=search_key_for_post, min_score=20)
94
95 # Loop through the returned posts and add a item for each to
96 # the list of results for Alfred
97 for post in posts:
98     wf.add_item(title=post['description'],
99                 subtitle=post['href'],
100                 arg=post['href'],
101                 valid=True,
102                 icon=ICON_WEB)
103
104 # Send the results to Alfred as XML
105 wf.send_feedback()
106 return 0
107
108
109 if __name__ == u"__main__":
110     wf = Workflow()
111     sys.exit(wf.run(main))

```

Quite a lot has happened here: at the top in line 5, we're importing a couple more icons that we use in `main()` to notify the user that their API key is missing and that they should set it (lines 65–72).

(You can see a list of all supported icons [here](#).)

We've adapted `get_recent_posts()` to accept an `api_key` argument. We *could* continue to use the `API_KEY` global variable, but that'd be bad form.

As a result of this, we've had to alter the way `Workflow.cached_data()` is called. It can't call a function that requires any arguments, so we've added a `wrapper()` function within `main()` (lines 82–87) that calls `get_recent_posts()` with the necessary `api_key` arguments, and we pass this `wrapper()` function (which needs no arguments) to `Workflow.cached_data()` instead (line 89).

At the top of `main()` (lines 39–49), we've added an argument parser using `argparse` that can take an optional `--apikey APIKEY` argument and an optional `query` argument (remember the script doesn't require a query).

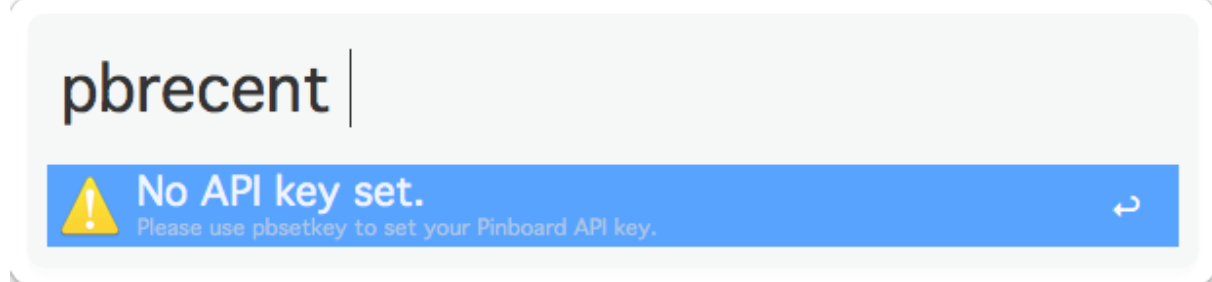
Then, in lines 55–59, we check if an API key was passed using `--apikey`. If it was, we save it using `settings` (see [below](#)).

Once this is done, we exit the script.

If no API key was specified with `--apikey`, we try to show/filter Pinboard posts as before. But first of all, we now have to check to see if we already have an API key saved (lines 65–72). If not, we show the user a warning (No API key set) and exit the script.

Finally, if we have an API key saved, we retrieve it and show/filter the Pinboard posts just as before (lines 78–107).

Of course, we don't have an API key saved, and we haven't yet set up our workflow in Alfred to save one, so the workflow currently won't work. Try to run it, and you'll see the warning we just implemented:



So let's add that functionality now.

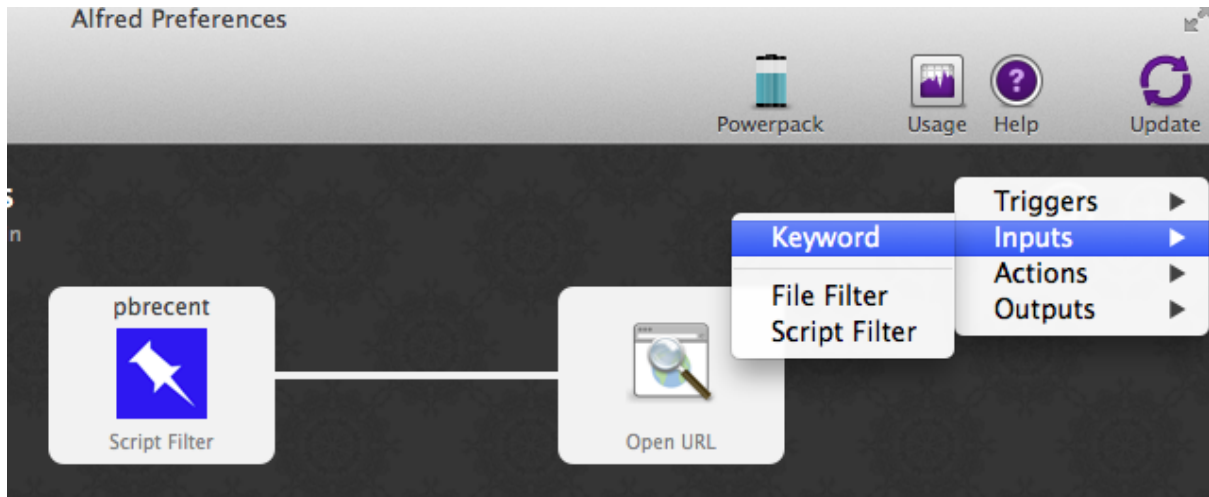
## Multi-step actions

Asking the user for input and saving it is best done in two steps:

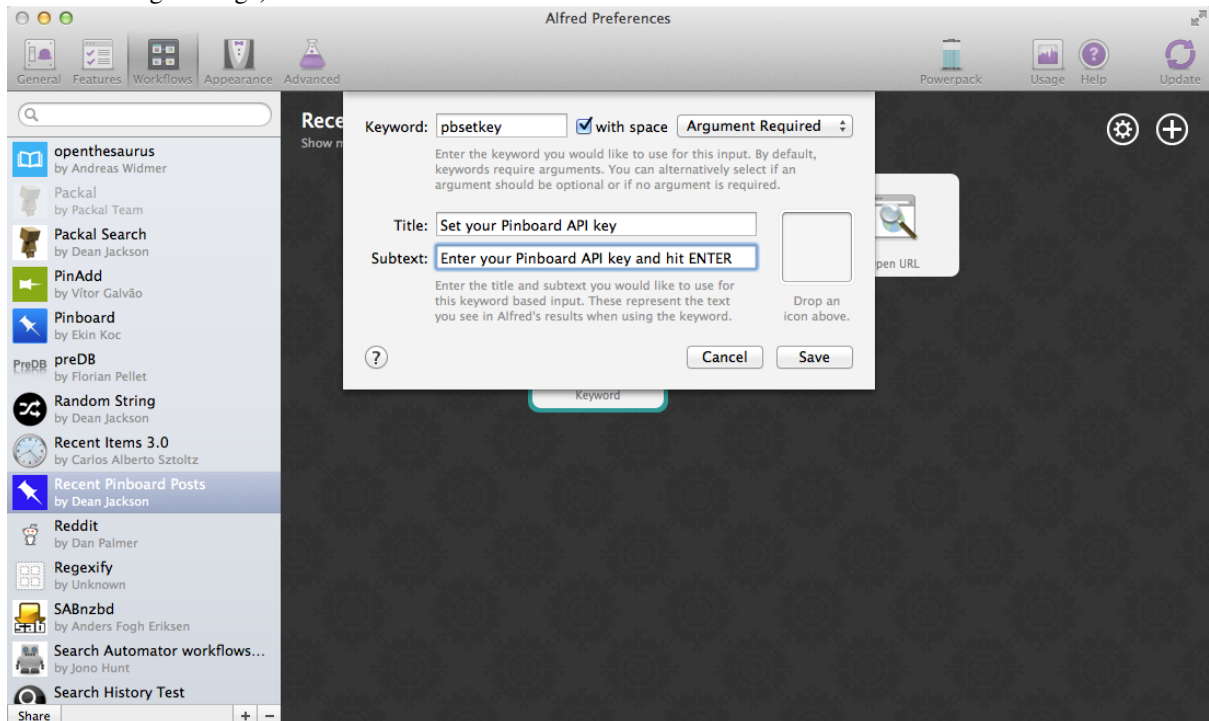
1. Ask for the data.
2. Pass it to a second action to save it.

A Script Filter is designed to be called constantly by Alfred and return results. This time, we just want to get some data, so we'll use a **Keyword** input instead.

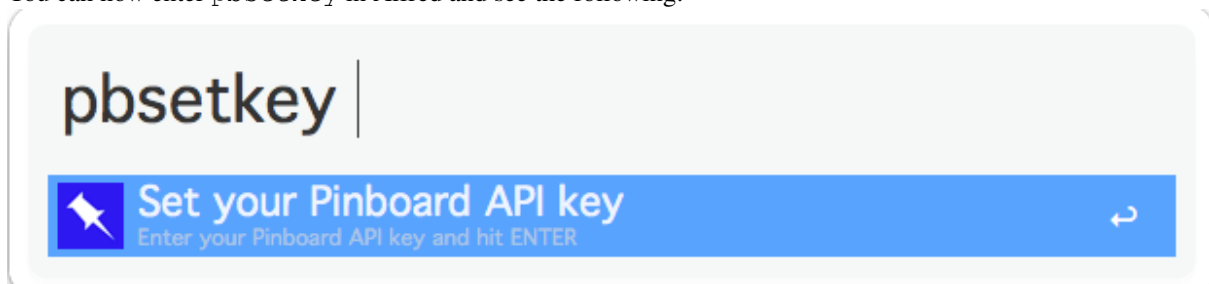
Go back to your workflow in Alfred's Preferences and add a **Keyword** input:



And set it up as follows (we'll use the keyword `pbsetkey` because that's what we told the user to use in the above warning message):

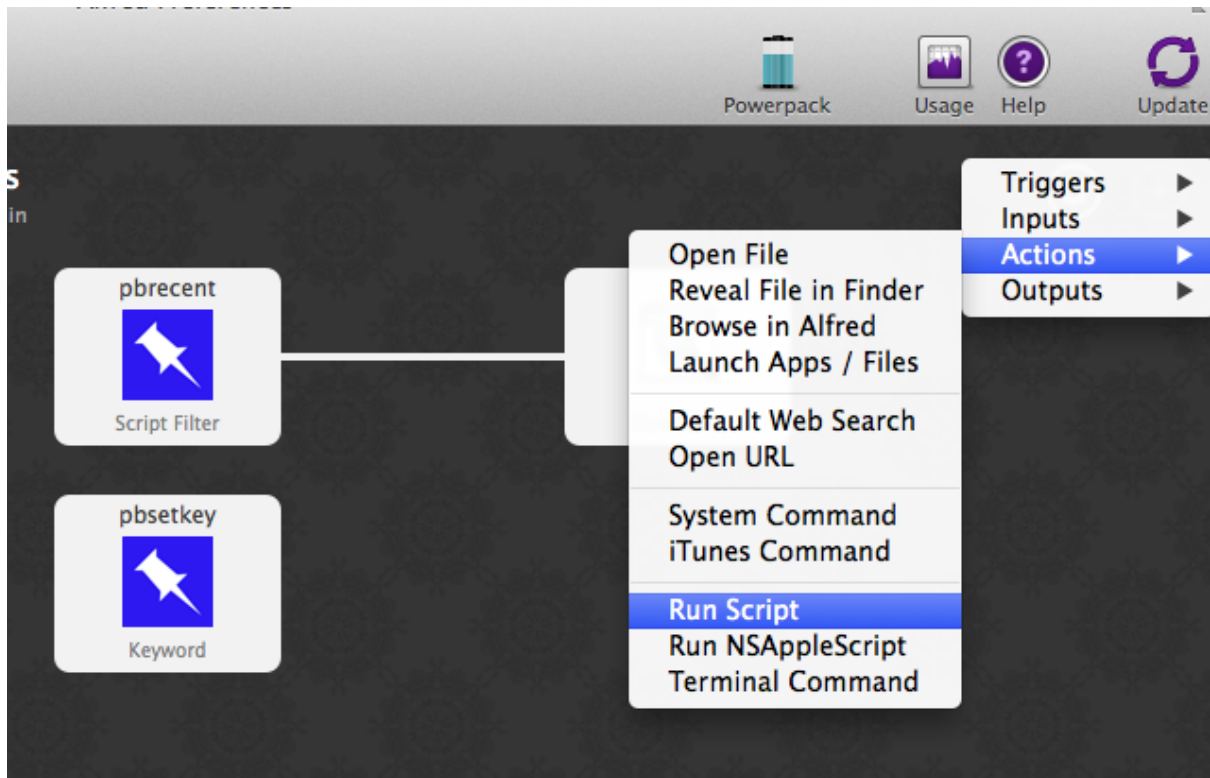


You can now enter `pbsetkey` in Alfred and see the following:

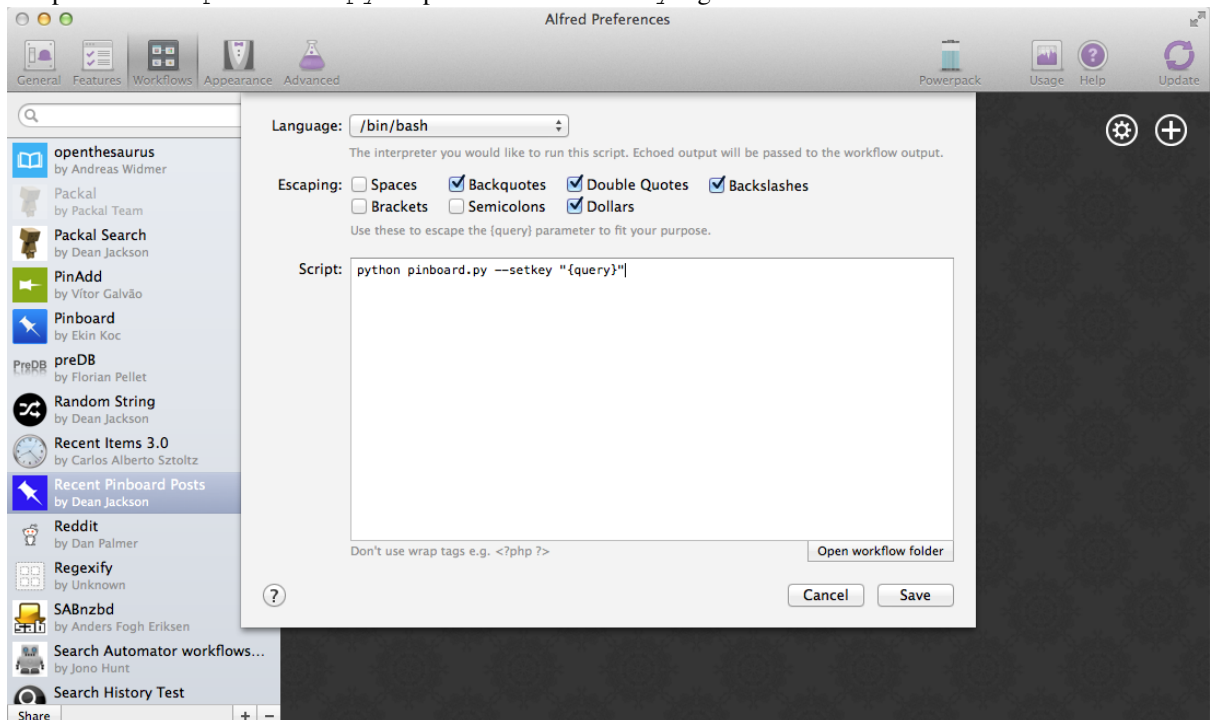


It won't do anything yet, though, as we haven't connected its output to anything.

Back in Alfred's Preferences, add a **Run Script** action:

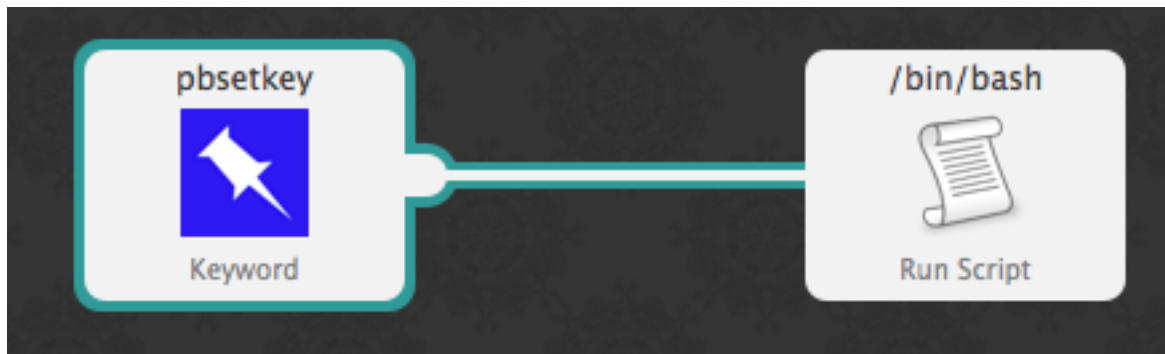


and point it at our `pinboard.py` script with the `--setkey` argument:



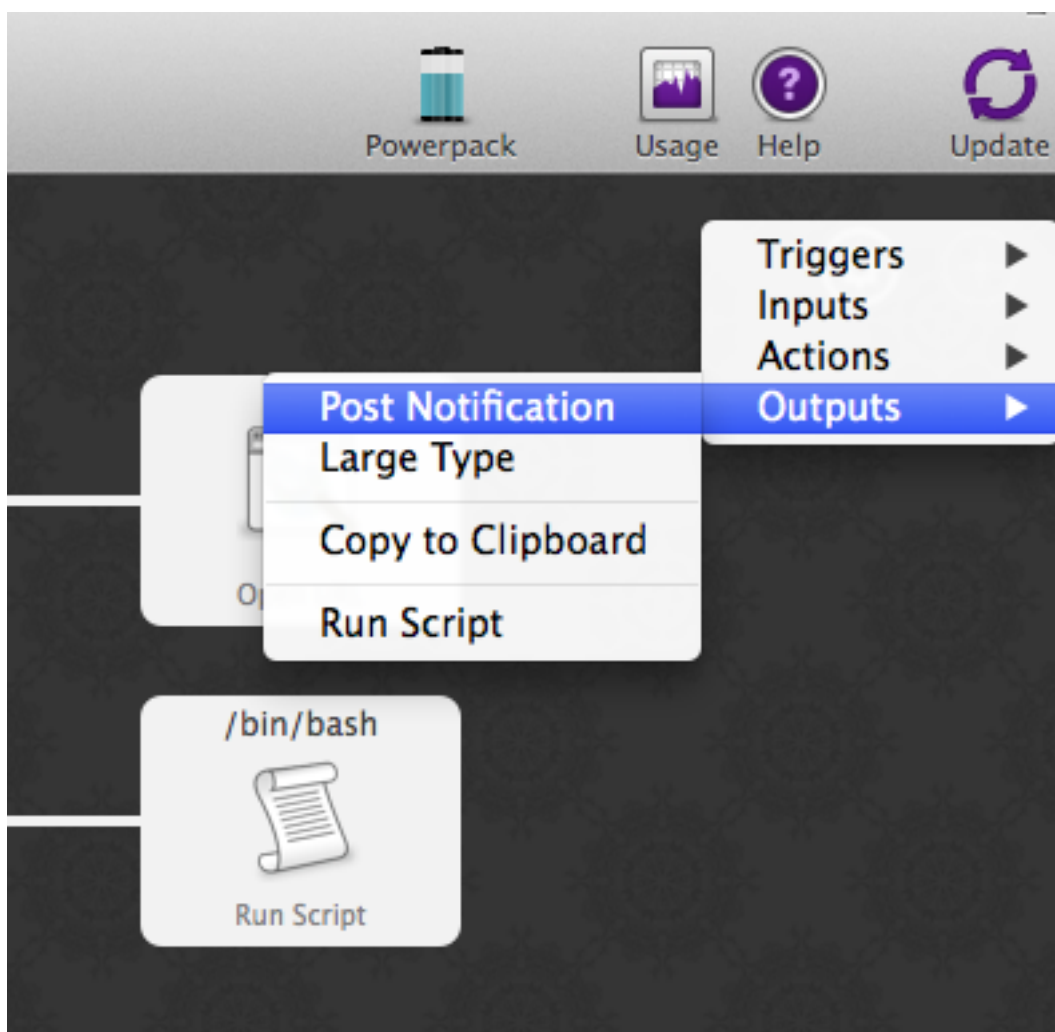
Finally, connect the `pbsetkey` **Keyword** to the new **Run Script** action:





Now you can call `pbsetkey` in Alfred, paste in your Pinboard API key and hit **ENTER**. It will be saved by the workflow and `pbrecent` will once again work as expected. Try it.

It's a little confusing receiving no feedback on whether the key was saved or not, so go back into Alfred's Preferences, and add an **Output > Post Notification** action to your workflow:



In the resulting pop-up, enter a message to be shown in Notification Center:

Output to: **Default (Notification Center)**

Select where you would like to post this notification to. It is best to leave this as default and set Alfred's global notification output.

☐ Only show if passed in argument has content

Only show this notification if {query} isn't empty.


Title: **Saved API key**

Text: **Your Pinboard API key was saved**

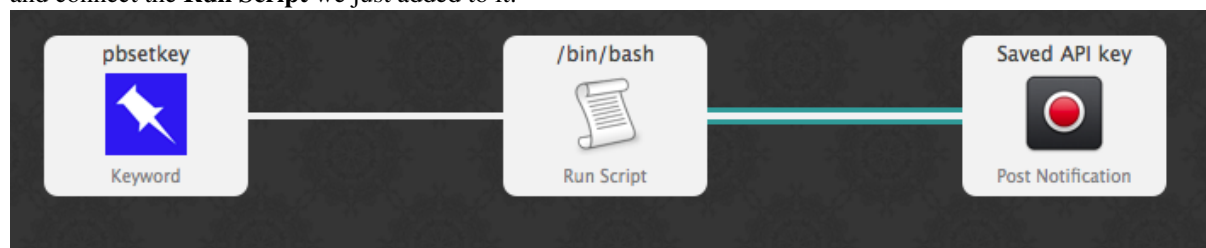
The title and text you would like to output. Use the {query} placeholder to output the argument.

☐ Last path component ☐ Remove extension

Use {query} to display the passed argument. You can do basic modifications to the {query}, useful if a filename. Use Last path component for only the filename and also remove the extension.

 **Cancel** **Save**

and connect the **Run Script** we just added to it:



Try setting your API key again with `pbsetkey` and this time you'll get a notification that it was saved.

### Saving settings

Saving the API key was pretty easy (1 line of code). `Settings` is a special dictionary that automatically saves itself when you change its contents. It can be used much like a normal dictionary with the caveat that all values must be serializable to JSON as the settings are saved as a JSON file in the workflow's data directory.

Very simple, yes, but secure? No. A better place to save the API key would be in the user's Keychain. Let's do that.

**Saving settings securely** *Workflow* provides three methods for managing data saved in OS X's Keychain: `get_password()`, `save_password()` and `delete_password()`.

They are all called with an `account` name and an optional `service` name (by default, this is your workflow's `bundle ID`).

Change your `pinboard.py` script as follows to use Keychain instead of a JSON file to store your API key:

```

1  # encoding: utf-8
2
3  import sys
4  import argparse
5  from workflow import Workflow, ICON_WEB, ICON_WARNING, web, PasswordNotFound
6
7
8  def get_recent_posts(api_key):
9      """Retrieve recent posts from Pinboard.in
10
11     Returns a list of post dictionaries.
12
13     """
14     url = 'https://api.pinboard.in/v1/posts/recent'
15     params = dict(auth_token=api_key, count=100, format='json')
16     r = web.get(url, params)
17
18     # throw an error if request failed
19     # Workflow will catch this and show it to the user
20     r.raise_for_status()
21
22     # Parse the JSON returned by pinboard and extract the posts
23     result = r.json()
24     posts = result['posts']
25     return posts
26
27
28  def search_key_for_post(post):
29      """Generate a string search key for a post"""
30     elements = []
31     elements.append(post['description']) # title of post
32     elements.append(post['tags']) # post tags
33     elements.append(post['extended']) # description
34     return u' '.join(elements)
35
36
37  def main(wf):
38
39     # build argument parser to parse script args and collect their
40     # values
41     parser = argparse.ArgumentParser()
42     # add an optional (nargs='?') --apikey argument and save its
43     # value to 'apikey' (dest). This will be called from a separate "Run Script"
44     # action with the API key
45     parser.add_argument('--setkey', dest='apikey', nargs='?', default=None)
46     # add an optional query and save it to 'query'
47     parser.add_argument('query', nargs='?', default=None)
48     # parse the script's arguments
49     args = parser.parse_args(wf.args)
50
51     #####
52     # Save the provided API key
53     #####
54
55     # decide what to do based on arguments
56     if args.apikey: # Script was passed an API key
57         # save the key
58         wf.save_password('pinboard_api_key', args.apikey)
59         return 0 # 0 means script exited cleanly
60
61     #####

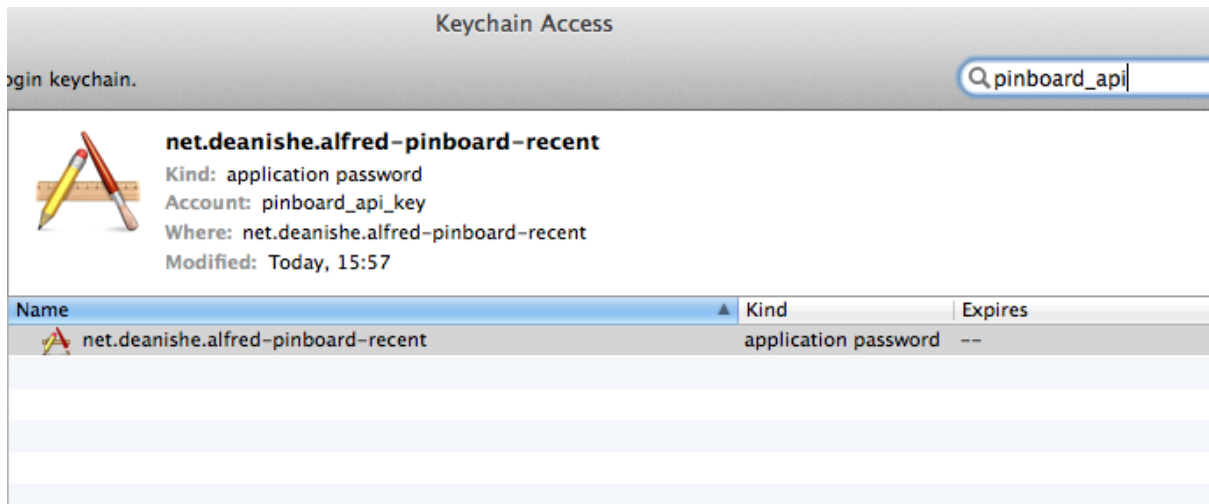
```

```
62  # Check that we have an API key saved
63  #####
64
65  try:
66      api_key = wf.get_password('pinboard_api_key')
67  except PasswordNotFound: # API key has not yet been set
68      wf.add_item('No API key set.',
69                  'Please use pbsetkey to set your Pinboard API key.',
70                  valid=False,
71                  icon=ICON_WARNING)
72      wf.send_feedback()
73      return 0
74
75  #####
76  # View/filter Pinboard posts
77  #####
78
79  query = args.query
80  # Retrieve posts from cache if available and no more than 600
81  # seconds old
82
83  def wrapper():
84      """`cached_data` can only take a bare callable (no args),
85      so we need to wrap callables needing arguments in a function
86      that needs none.
87      """
88      return get_recent_posts(api_key)
89
90  posts = wf.cached_data('posts', wrapper, max_age=600)
91
92  # If script was passed a query, use it to filter posts
93  if query:
94      posts = wf.filter(query, posts, key=search_key_for_post, min_score=20)
95
96  # Loop through the returned posts and add an item for each to
97  # the list of results for Alfred
98  for post in posts:
99      wf.add_item(title=post['description'],
100                  subtitle=post['href'],
101                  arg=post['href'],
102                  valid=True,
103                  icon=ICON_WEB)
104
105  # Send the results to Alfred as XML
106  wf.send_feedback()
107  return 0
108
109
110  if __name__ == u"__main__":
111      wf = Workflow()
112      sys.exit(wf.run(main))
```

`get_password()` raises a `PasswordNotFound` exception if the requested password isn't in your Keychain, so we import `PasswordNotFound` and change `if not api_key:` to a `try ... except` clause (lines 65–72).

Try running your workflow again. It will complain that you haven't saved your API key (it's looking in Keychain now, not the settings), so set your API key once again, and you should be able to browse your recent posts in Alfred once more.

And if you open **Keychain Access**, you'll find the API key safely tucked away in your Keychain:

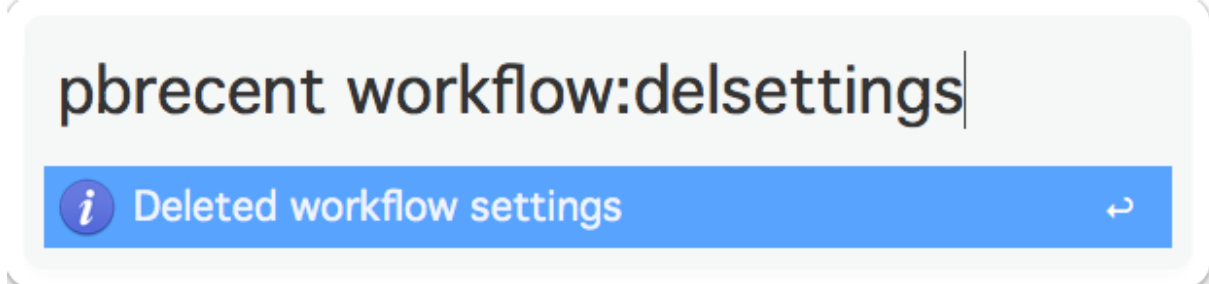


As a bonus, if you have multiple Macs and use iCloud Keychain, the API key will be seamlessly synced across machines, saving you the trouble of setting up the workflow multiple times.

### “Magic” arguments

Now that the API key is stored in Keychain, we don’t need it saved in the workflow’s settings any more (and having it there that kind of defeats the purpose of using Keychain). To get rid of it, we can use one of Alfred-Workflow’s “magic” arguments: `workflow:delsettings`.

Open up Alfred, and enter `pbrecent workflow:delsettings`. You should see the following message:



Alfred-Workflow has recognised one of its “magic” arguments, performed the corresponding action, logged it to the log file, notified the user via Alfred and exited the workflow.

Magic arguments are designed to help coders develop and debug workflows. See *“Magic” arguments* for more details.

### Logging

There’s a log, you say? Yup. There’s a `logging.Logger` instance at `Workflow.logger` configured to output to both the Terminal (in case you’re running your workflow script in Terminal) and your workflow’s log file. Normally, I use it like this:

```

1  from workflow import Workflow
2
3  log = None
4
5
6  def main(wf):
7      log.debug('Started')
8
9  if __name__ == '__main__':
10     wf = Workflow()

```

```
11     log = wf.logger
12     wf.run(main)
```

Assigning `Workflow.logger` to the module-global `log` means it can be accessed from within any function without having to pass the `Workflow` or `Workflow.logger` instance around.

The `wf` object is also a module-level global, but it's only created if the script is run, not imported.

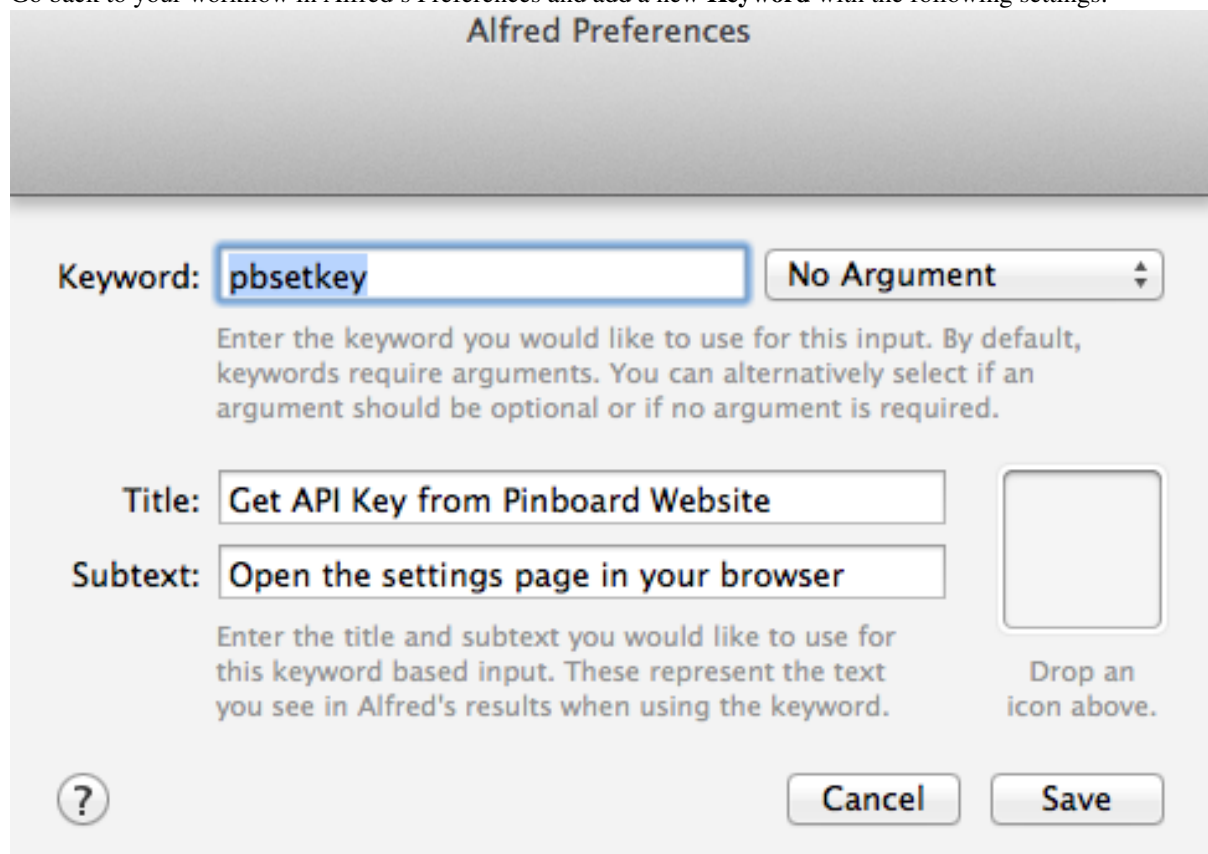
### Spit and polish

So far, the workflow's looking pretty good. But there are still a few of things that could be better. For one, it's not necessarily obvious to a user where to find their Pinboard API key (it took me a good, hard Googling to find it while writing these tutorials). For another, it can be confusing if there are no results from a workflow and Alfred shows its default Google/Amazon searches instead. Finally, the workflow is unresponsive while updating the list of recent posts from Pinboard. That can't be helped if we don't have any posts cached, but apart from the very first run, we always will, so why don't we show what we have and update in the background?

Let's fix those issues. The easy ones first.

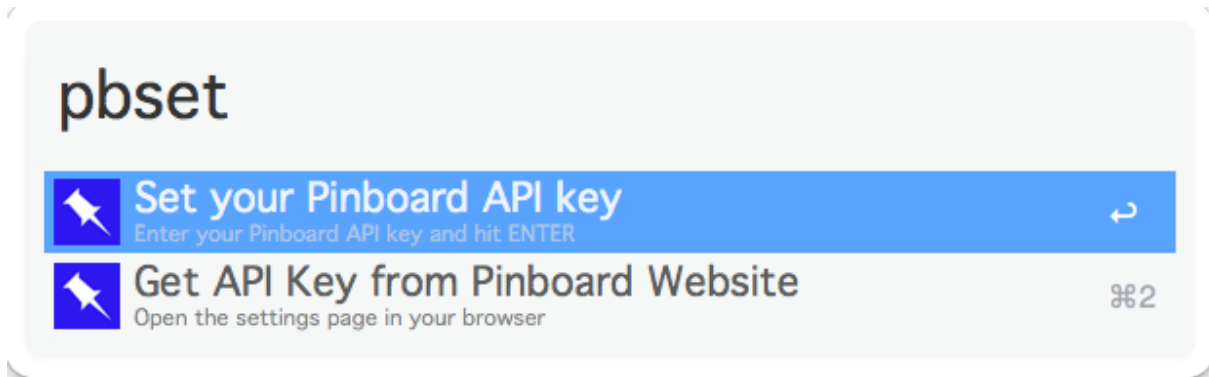
**Two actions, one keyword** To solve the first issue (Pinboard API keys being hard to find), we'll add a second **Keyword** input that responds to the same `pbsetkey` keyword as our other action, but this one will just send the user to the Pinboard [password settings page](#) where the API keys are kept.

Go back to your workflow in Alfred's Preferences and add a new **Keyword** with the following settings:



The screenshot shows the 'Alfred Preferences' window. Under the 'Keywords' tab, a new keyword is being configured. The 'Keyword' field contains 'pbsetkey' and the 'No Argument' dropdown is selected. Below this is a descriptive text: 'Enter the keyword you would like to use for this input. By default, keywords require arguments. You can alternatively select if an argument should be optional or if no argument is required.' The 'Title' field contains 'Get API Key from Pinboard Website' and the 'Subtext' field contains 'Open the settings page in your browser'. Below these fields is another descriptive text: 'Enter the title and subtext you would like to use for this keyword based input. These represent the text you see in Alfred's results when using the keyword.' To the right of the title and subtext fields is a large empty box with the text 'Drop an icon above.' at the bottom. At the bottom left is a help icon (a circle with a question mark). At the bottom right are 'Cancel' and 'Save' buttons.

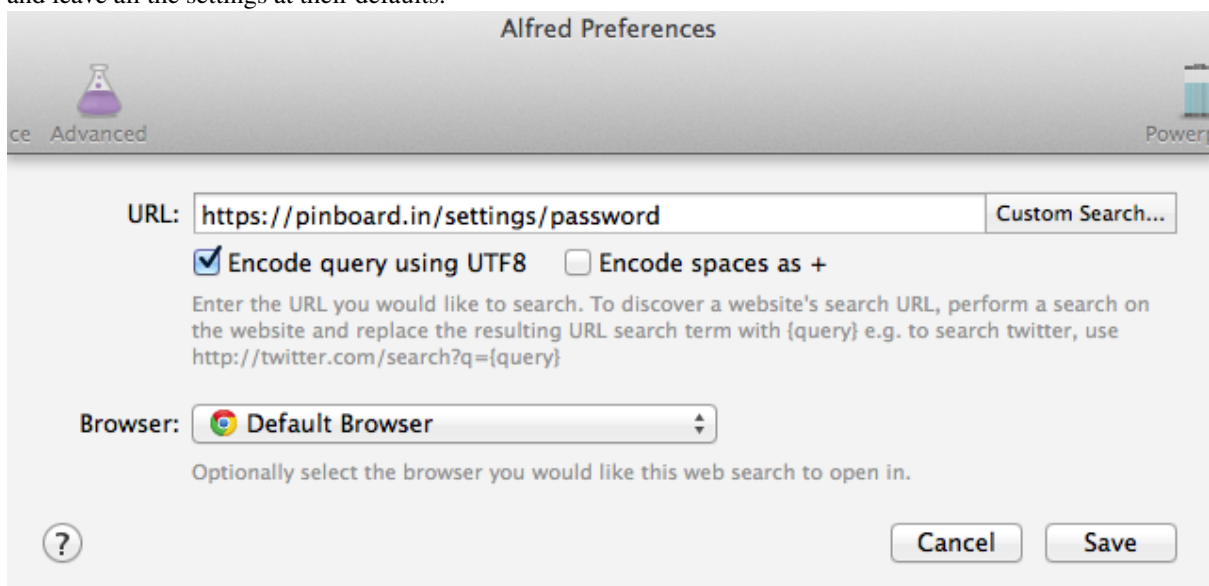
Now when you type `pbsetkey` into Alfred, you should see two options:



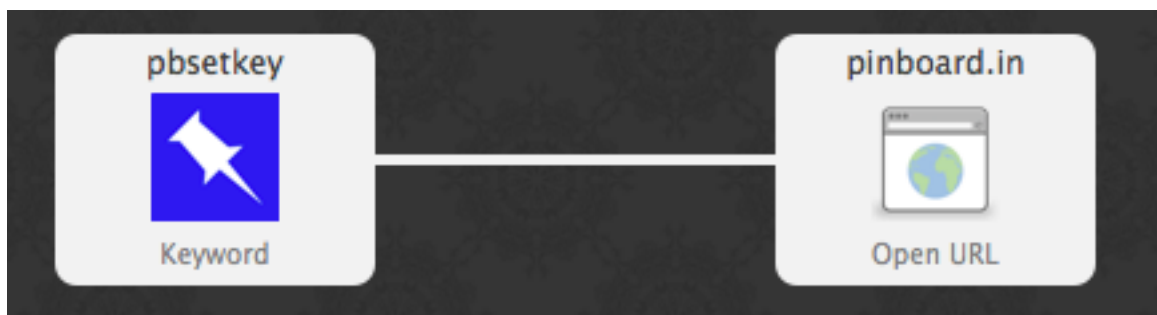
The second action doesn't do anything yet, of course, because we haven't connected it to anything. So add an **Open URL** action in Alfred, enter this URL:

<https://pinboard.in/settings/password>

and leave all the settings at their defaults.



Finally, connect your new **Keyword** to the new **Open URL** action:



Enter `pbsetkey` into Alfred once more and try out the new action. Pinboard should open in your default browser. Easy peasy.

**Notifying the user if there are no results** Alfred's default behaviour when a Script Filter returns no results is to show its fallback searches. This is also what it does if a workflow crashes. So, the best thing to do when a user is explicitly using your workflow is to show a message indicating that no results were found.

Change `pinboard.py` to the following:

```

1  # encoding: utf-8
2
3  import sys
4  import argparse
5  from workflow import Workflow, ICON_WEB, ICON_WARNING, web, PasswordNotFound
6
7
8  def get_recent_posts(api_key):
9      """Retrieve recent posts from Pinboard.in
10
11     Returns a list of post dictionaries.
12
13     """
14     url = 'https://api.pinboard.in/v1/posts/recent'
15     params = dict(auth_token=api_key, count=100, format='json')
16     r = web.get(url, params)
17
18     # throw an error if request failed
19     # Workflow will catch this and show it to the user
20     r.raise_for_status()
21
22     # Parse the JSON returned by pinboard and extract the posts
23     result = r.json()
24     posts = result['posts']
25     return posts
26
27
28  def search_key_for_post(post):
29      """Generate a string search key for a post"""
30     elements = []
31     elements.append(post['description']) # title of post
32     elements.append(post['tags']) # post tags
33     elements.append(post['extended']) # description
34     return u' '.join(elements)
35
36
37  def main(wf):
38
39     # build argument parser to parse script args and collect their
40     # values
41     parser = argparse.ArgumentParser()
42     # add an optional (nargs='?') --apikey argument and save its
43     # value to 'apikey' (dest). This will be called from a separate "Run Script"
44     # action with the API key
45     parser.add_argument('--apikey', dest='apikey', nargs='?', default=None)
46     # add an optional query and save it to 'query'
47     parser.add_argument('query', nargs='?', default=None)
48     # parse the script's arguments
49     args = parser.parse_args(wf.args)
50
51     #####
52     # Save the provided API key
53     #####
54
55     # decide what to do based on arguments
56     if args.apikey: # Script was passed an API key
57         # save the key
58         wf.save_password('pinboard_api_key', args.apikey)
59         return 0 # 0 means script exited cleanly
60
61     #####
62     # Check that we have an API key saved
63     #####

```



```

64
65     try:
66         api_key = wf.get_password('pinboard_api_key')
67     except PasswordNotFound: # API key has not yet been set
68         wf.add_item('No API key set.',
69                     'Please use pbsetkey to set your Pinboard API key.',
70                     valid=False,
71                     icon=ICON_WARNING)
72         wf.send_feedback()
73         return 0
74
75     #####
76     # View/filter Pinboard posts
77     #####
78
79     query = args.query
80     # Retrieve posts from cache if available and no more than 600
81     # seconds old
82
83     def wrapper():
84         """`cached_data` can only take a bare callable (no args),
85         so we need to wrap callables needing arguments in a function
86         that needs none.
87         """
88         return get_recent_posts(api_key)
89
90     posts = wf.cached_data('posts', wrapper, max_age=600)
91
92     # If script was passed a query, use it to filter posts
93     if query:
94         posts = wf.filter(query, posts, key=search_key_for_post, min_score=20)
95
96     if not posts: # we have no data to show, so show a warning and stop
97         wf.add_item('No posts found', icon=ICON_WARNING)
98         wf.send_feedback()
99         return 0
100
101     # Loop through the returned posts and add an item for each to
102     # the list of results for Alfred
103     for post in posts:
104         wf.add_item(title=post['description'],
105                     subtitle=post['href'],
106                     arg=post['href'],
107                     valid=True,
108                     icon=ICON_WEB)
109
110     # Send the results to Alfred as XML
111     wf.send_feedback()
112     return 0
113
114
115 if __name__ == u"__main__":
116     wf = Workflow()
117     sys.exit(wf.run(main))

```

In lines 96-99, we check to see if there are any posts, and if not, we show the user a warning, send the results to Alfred and exit. This does away with Alfred's distracting default searches and lets the user know exactly what's going on.

**Greased lightning: background updates** All that remains is for our workflow to provide the blazing fast results Alfred users have come to expect. No waiting around for glacial web services for the likes of us. As long as we have some posts saved in the cache, we can show those while grabbing an updated list in the background (and

notifying the user of the update, of course).

Now, there are a few different ways to start a background process. We could ask the user to set up a `cron` job, but `cron` isn't the easiest software to use. We could add and load a [Launch Agent](#), but that'd run indefinitely, whether or not the workflow is being used, and even if the workflow were uninstalled. So we'd best start our background process from within the workflow itself.

Normally, you'd use `subprocess.Popen` to start a background process, but that doesn't work quite as you might expect in Alfred: it treats your workflow as still running till the background process has finished, too, so it won't call your workflow with a new query till the update is done. Which is exactly what happens now and the behaviour we want to avoid.

Fortunately, Alfred-Workflow provides the `background` module to solve this problem.

Using the `background.run_in_background()` and `background.is_running()` functions, we can easily run a script in the background while our workflow remains responsive to Alfred's queries.

Alfred-Workflow's `background` module is based on, and uses the same API as `subprocess.call()`, but it runs the command as a background process (consequently, it won't return anything). So, our updater script will be called from our main workflow script, but `background` will run it as a background process. This way, it will appear to exit immediately, so Alfred will keep on calling our workflow every time the query changes.

Meanwhile, our main workflow script will check if the background updater is running and post a useful, friendly notification if it is.

Let's have at it.

**Background updater script** Create a new file in the workflow root directory called `update.py` with these contents:

```
1  # encoding: utf-8
2
3
4  from workflow import web, Workflow, PasswordNotFound
5
6
7  def get_recent_posts(api_key):
8      """Retrieve recent posts from Pinboard.in
9
10     Returns a list of post dictionaries.
11
12     """
13     url = 'https://api.pinboard.in/v1/posts/recent'
14     params = dict(auth_token=api_key, count=100, format='json')
15     r = web.get(url, params)
16
17     # throw an error if request failed
18     # Workflow will catch this and show it to the user
19     r.raise_for_status()
20
21     # Parse the JSON returned by pinboard and extract the posts
22     result = r.json()
23     posts = result['posts']
24     return posts
25
26
27  def main(wf):
28      try:
29          # Get API key from Keychain
30          api_key = wf.get_password('pinboard_api_key')
31
32          # Retrieve posts from cache if available and no more than 600
33          # seconds old
34
```

```

35     def wrapper():
36         """`cached_data` can only take a bare callable (no args),
37         so we need to wrap callables needing arguments in a function
38         that needs none.
39         """
40         return get_recent_posts(api_key)
41
42     posts = wf.cached_data('posts', wrapper, max_age=600)
43     # Record our progress in the log file
44     wf.logger.debug('{} Pinboard posts cached'.format(len(posts)))
45
46     except PasswordNotFound: # API key has not yet been set
47         # Nothing we can do about this, so just log it
48         wf.logger.error('No API key saved')
49
50 if __name__ == '__main__':
51     wf = Workflow()
52     wf.run(main)

```

At the top of the file (line 7), we've copied the `get_recent_posts()` function from `pinboard.py` (we won't need it there any more).

The contents of the `try` block in `main()` (lines 29–44) are once again copied straight from `pinboard.py` (where we won't be needing them any more).

The `except` clause (lines 46–48) is to trap the `PasswordNotFound` error that `Workflow.get_password()` will raise if the user hasn't set their API key via Alfred yet. `update.py` can quietly die if no API key has been set because `pinboard.py` takes care of notifying the user to set their API key.

Let's try out `update.py`. Open a Terminal window at the workflow root directory and run the following:

```
python update.py
```

If it works, you should see something like this:

```

1  21:59:59 workflow.py:855 DEBUG    get_password : net.deanishe.alfred-pinboard-recent:pinboard_ap
2  21:59:59 workflow.py:544 DEBUG    Loading cached data from : /Users/dean/Library/Caches/com.runn
3  21:59:59 update.py:111 DEBUG      100 Pinboard posts cached
4  22:19:25 workflow.py:371 INFO     Opening workflow log file

```

As you can see in the 3rd line, `update.py` did its job.

**Running `update.py` from `pinboard.py`** So now let's update `pinboard.py` to call `update.py` instead of doing the update itself:

```

1  # encoding: utf-8
2
3  import sys
4  import argparse
5  from workflow import (Workflow, ICON_WEB, ICON_INFO, ICON_WARNING,
6                        PasswordNotFound)
7  from workflow.background import run_in_background, is_running
8
9
10 def search_key_for_post(post):
11     """Generate a string search key for a post"""
12     elements = []
13     elements.append(post['description']) # title of post
14     elements.append(post['tags']) # post tags
15     elements.append(post['extended']) # description
16     return u' '.join(elements)
17

```

```

18
19 def main(wf):
20
21     # build argument parser to parse script args and collect their
22     # values
23     parser = argparse.ArgumentParser()
24     # add an optional (nargs='?') --apikey argument and save its
25     # value to 'apikey' (dest). This will be called from a separate "Run Script"
26     # action with the API key
27     parser.add_argument('--setkey', dest='apikey', nargs='?', default=None)
28     # add an optional query and save it to 'query'
29     parser.add_argument('query', nargs='?', default=None)
30     # parse the script's arguments
31     args = parser.parse_args(wf.args)
32
33     #####
34     # Save the provided API key
35     #####
36
37     # decide what to do based on arguments
38     if args.apikey: # Script was passed an API key
39         # save the key
40         wf.save_password('pinboard_api_key', args.apikey)
41         return 0 # 0 means script exited cleanly
42
43     #####
44     # Check that we have an API key saved
45     #####
46
47     try:
48         wf.get_password('pinboard_api_key')
49     except PasswordNotFound: # API key has not yet been set
50         wf.add_item('No API key set.',
51                     'Please use pbsetkey to set your Pinboard API key.',
52                     valid=False,
53                     icon=ICON_WARNING)
54         wf.send_feedback()
55         return 0
56
57     #####
58     # View/filter Pinboard posts
59     #####
60
61     query = args.query
62
63     # Get posts from cache. Set `data_func` to None, as we don't want to
64     # update the cache in this script and `max_age` to 0 because we want
65     # the cached data regardless of age
66     posts = wf.cached_data('posts', None, max_age=0)
67
68     # Start update script if cached data is too old (or doesn't exist)
69     if not wf.cached_data_fresh('posts', max_age=600):
70         cmd = ['/usr/bin/python', wf.workflowfile('update.py')]
71         run_in_background('update', cmd)
72
73     # Notify the user if the cache is being updated
74     if is_running('update'):
75         wf.add_item('Getting new posts from Pinboard',
76                     valid=False,
77                     icon=ICON_INFO)
78
79     # If script was passed a query, use it to filter posts if we have some
80     if query and posts:

```

```

81     posts = wf.filter(query, posts, key=search_key_for_post, min_score=20)
82
83     if not posts: # we have no data to show, so show a warning and stop
84         wf.add_item('No posts found', icon=ICON_WARNING)
85         wf.send_feedback()
86         return 0
87
88     # Loop through the returned posts and add a item for each to
89     # the list of results for Alfred
90     for post in posts:
91         wf.add_item(title=post['description'],
92                     subtitle=post['href'],
93                     arg=post['href'],
94                     valid=True,
95                     icon=ICON_WEB)
96
97     # Send the results to Alfred as XML
98     wf.send_feedback()
99     return 0
100
101
102 if __name__ == u"__main__":
103     wf = Workflow()
104     sys.exit(wf.run(main))

```

First of all, we’ve changed the imports a bit. We no longer need `workflow.web`, because we’ll use the functions `run_in_background()` from `workflow.background` to call `update.py` instead, and we’ve also imported another icon (`ICON_INFO`) to show our update message.

As noted before, `get_recent_posts()` has now moved to `update.py`, as has the `wrapper()` function inside `main()`.

Also in `main()`, we no longer need `api_key`. However, we still want to know if it has been saved, so we can show a warning if not, so we still call `Workflow.get_password()`, but without saving the result.

Most importantly, we’ve now expanded the update code to check if our cached data is fresh with `Workflow.cached_data_fresh()` and to run the `update.py` script via `background.run_in_background()` if not (`Workflow.workflowfile()` returns the full path to a file in the workflow’s root directory).

Then we check if the update process is running via `background.is_running()` using the name we assigned to the process (`update`), and notify the user via Alfred’s results if it is.

Finally, we call `Workflow.cached_data()` with `None` as the data-retrieval function (line 66) because we don’t want to run an update from this script, blocking Alfred. As a consequence, it’s possible that we’ll get back `None` instead of a list of posts if there are no cached data, so we check for this before trying to filter `None` in line 80.

### The fruits of your labour

Now let’s give it a spin. Open up Alfred and enter `pbrecent workflow:delcache` to clear the cached data. Then enter `pbrecent` and start typing a query. You should see the “Getting new posts from Pinboard” message appear. Unfortunately, we won’t see any results at the moment because we just deleted the cached data.

To see our background updater weave its magic, we can change the `max_age` parameter passed to `Workflow.cached_data()` in `update.py` on line 42 and to `Workflow.cached_data_fresh()` in `pinboard.py` on line 69 to 60. Open up Alfred, enter `pbrecent` and a couple of letters, then twiddle your thumbs for ~55 seconds. Type another letter or two and you should see the “Getting new posts…” message and search results. Cool, huh?

**Sharing your workflow** Now you’ve produced a technical marvel, it’s time to tell the world and enjoy the well-earned plaudits. To build your workflow, open it up in Alfred’s Preferences, right-click on the workflow’s name in the list on the left-hand side, and choose **Export...** This will save a `.alfredworkflow` file that you can share with other people. `.alfredworkflow` files are just ZIP files with a different extension. If you want to have a poke around inside one, just change the extension to `.zip` and extract it in the normal way.

And how do you share your Workflow with the world?

There’s a [Share your Workflows thread on the official Alfred forum](#), but being a forum, it’s less than ideal as a directory for workflows. Also, you’d need to find your own place to host your workflow file (for which GitHub and Dropbox are both good, free choices).

It’s a good idea to sign up for the Alfred forum and post a thread for your workflow, so users can get in touch with you, but you might want to consider uploading it to [Packal.org](#), a site specifically designed for hosting Alfred workflows. Your workflow will be much easier to find on that site than in the forum, and they’ll also host the workflow download for you.

**Updating your workflow** Software, like plans, never survives contact with the enemy, err, user.

It’s likely that a bug or two will be found and some sweet improvements will be suggested, and so you’ll probably want to release a new and improved version of your workflow somewhere down the line.

Instead of requiring your users to regularly visit a forum thread or a website to check for an update, there are a couple of ways you can have your workflow (semi-)automatically updated.

**The Packal Updater** The simplest way in terms of implementation is to upload your workflow to [Packal.org](#). If you release a new version, any user who also uses the [Packal Updater workflow](#) will then be notified of the updated version. The disadvantage of this method is it only works if a user installs and uses the [Packal Updater workflow](#).

**GitHub releases** A *slightly* more complex to implement method is to use Alfred-Workflow’s built-in support for updates via [GitHub releases](#). If you tell your `Workflow` object the name of your GitHub repo and the installed workflow’s version number, Alfred-Workflow will automatically check for a new version every day.

By default, Alfred-Workflow won’t inform the user of the new version or update the workflow unless the user explicitly uses the `workflow:update` “*magic*” *argument*, but you can check the `Workflow.update_available` attribute and inform the user of the availability of an update if it’s `True`.

See [Self-updating](#) in the *User Manual* for information on how to enable your workflow to update itself from GitHub.

---

## User Manual

---

If you know your way around Python and Alfred, here's an overview of what Alfred-Workflow can do and how to do it.

### 5.1 User Manual

This section describes how to use the features of Alfred-Workflow.

If you're new to writing workflows or coding in general, start with the [Tutorial](#).

---

**Tip:** If you're writing a workflow that uses data from the system (e.g. from files/the filesystem or via command-line programs called via `subprocess`), please read [Encoded strings and Unicode](#), which describes how to handle data from sources other than Alfred-Workflow's libraries.

---

#### 5.1.1 Workflow setup and skeleton

**Alfred-Python** is aimed particularly at authors of so-called **Script Filters**. These are activated by a keyword in Alfred, receive user input and return results to Alfred.

To write a Script Filter with Alfred-Workflow, make sure your Script Filter is set to use `/bin/bash` as the **Language**, and select the following (and only the following) **Escaping** options:

- Backquotes
- Double Quotes
- Dollars
- Backslashes

The **Script** field should contain the following:

```
python yourscrip.py "{query}"
```

where `yourscrip.py` is the name of your script.

Your workflow should start out like this. This enables `Workflow` to capture any errors thrown by your scripts:

```
1 #!/usr/bin/python
2 # encoding: utf-8
3
4 import sys
5
6 from workflow import Workflow
7
8 log = None
9
```

```
10
11 def main(wf):
12     # The Workflow instance will be passed to the function
13     # you call from `Workflow.run`
14
15     # Your imports here if you want to catch import errors
16     import somemodule
17     import anothermodule
18
19     # Get args from Workflow as normalized Unicode
20     args = wf.args
21
22     # Do stuff here ...
23
24     # Add an item to Alfred feedback
25     wf.add_item('Item title', 'Item subtitle')
26
27     # Send output to Alfred
28     wf.send_feedback()
29
30
31 if __name__ == '__main__':
32     wf = Workflow()
33     # Assign Workflow logger to a global variable, so all module
34     # functions can access it without having to pass the Workflow
35     # instance around
36     log = wf.logger
37     sys.exit(wf.run(main))
```

## 5.1.2 Including 3rd party libraries

It's a Very Bad Idea™ to install (or ask users to install) 3rd-party libraries in the OS X system Python. Alfred-Workflow makes it easy to include them in your Workflow.

Simply create a `lib` subdirectory under your Workflow's root directory and install your dependencies there. You can call the directory whatever you want, but in the following explanation, I'll assume you used `lib`.

To install libraries in your dependencies directory, use:

```
pip install --target=path/to/my/workflow/lib python-lib-name
```

The path you pass as the `--target` argument should be the path to the directory under your Workflow's root directory in which you want to install your libraries. `python-lib-name` should be the “pip name” (i.e. the name the library has on [PyPI](#)) of the library you want to install, e.g. `requests` or `feedparser`.

This name is usually, but not always, the same as the name you use with `import`.

For example, to install Alfred-Workflow, you would run `pip install Alfred-Workflow` but use `import workflow` to import it.

**An example:** You're in a shell in Terminal.app in the Workflow's root directory and you're using `lib` as the directory for your Python libraries. You want to install `requests`. You would run:

```
pip install --target=lib requests
```

This will install the `requests` library into the `lib` subdirectory of the current working directory.

Then you instantiate `Workflow` with the `libraries` argument:

```
1 from workflow import Workflow
2
3 def main(wf):
4     import requests # Imported from ./lib
5
```



```

6 if __name__ == '__main__':
7     wf = Workflow(libraries=['./lib'])
8     sys.exit(wf.run(main))

```

When using this feature you **do not** need to create an `__init__.py` file in the `lib` subdirectory. `Workflow(..., libraries=['./lib'])` and creating `./lib/__init__.py` are effectively equal alternatives.

Instead of using `Workflow(..., libraries=['./lib'])`, you can add an empty `__init__.py` file to your `lib` subdirectory and import the libraries installed therein using:

```
from lib import requests
```

instead of simply:

```
import requests
```

### 5.1.3 Persistent data

Alfred provides special data and cache directories for each Workflow (in `~/Library/Application Support` and `~/Library/Caches` respectively). *Workflow* provides the following attributes/methods to make it easier to access these directories:

- `datadir` — The full path to your Workflow’s data directory.
- `cachedir` — The full path to your Workflow’s cache directory.
- `datafile(filename)` — The full path to `filename` under the data directory.
- `cachefile(filename)` — The full path to `filename` under the cache directory.

The cache directory may be deleted during system maintenance, and is thus only suitable for temporary data or data that is easily recreated. *Workflow*’s cache methods reflect this, and make it easy to replace cached data that are too old. See *Caching data* for details of the data caching API.

The data directory is intended for more permanent, user-generated data, or data that cannot be otherwise easily recreated. See *Storing data* for details of the data storage API.

It is easy to specify a custom file format for your stored data via the `serializer` argument if you want your data to be readable by the user or by other software. See *Serialization of stored/cached data* for more details.

---

**Tip:** There are also similar methods related to the root directory of your Workflow (where `info.plist` and your code are):

- `workflowdir` — The full path to your Workflow’s root directory.
- `workflowfile(filename)` — The full path to `filename` under your Workflow’s root directory.

These are used internally to implement “*Magic*” arguments, which provide assistance with debugging, updating and managing your workflow.

---

In addition, *Workflow* also provides a convenient interface for storing persistent settings with *Workflow.settings*. See *Settings* and *Keychain access* for more information on storing settings and sensitive data.

### Caching data

*Workflow* provides a few methods to simplify caching data that is slow to retrieve or expensive to generate (e.g. downloaded from a web API). These data are cached in your workflow’s cache directory (see `cachedir`). The main method is *Workflow.cached\_data()*, which takes a name under which the data should be cached, a callable to retrieve the data if they aren’t in the cache (or are too old), and a maximum age in seconds for the cached data:

```
1 from workflow import web, Workflow
2
3 def get_data():
4     return web.get('https://example.com/api/stuff').json()
5
6 wf = Workflow()
7 data = wf.cached_data('stuff', get_data, max_age=600)
```

To retrieve data only if they are in the cache, call with `None` as the data-retrieval function (which is the default):

```
data = wf.cached_data('stuff', max_age=600)
```

---

**Note:** This will return `None` if there are no corresponding data in the cache.

---

This is useful if you want to update your cache in the background, so it doesn't impact your Workflow's responsiveness in Alfred. (See [the tutorial](#) for an example of how to run an update script in the background.)

---

**Tip:** Passing `max_age=0` will return the cached data regardless of age.

---

## Clearing cached data

There is a convenience method for clearing a workflow's cache directory.

`clear_cache()` will by default delete all the files contained in `cachedir`. This is the method called if you use the `workflow:delcache` or `workflow:reset` [magic arguments](#).

You can selectively delete files from the cache by passing the optional `filter_func` argument to `clear_cache()`. This callable will be called with the filename (not path) of each file in the workflow's cache directory.

If `filter_func` returns `True`, the file will be deleted, otherwise it will be left in the cache. For example, to delete all `.zip` files in the cache, use:

```
1 def myfilter(filename):
2     return filename.endswith('.zip')
3
4 wf.clear_cache(myfilter)
```

or more simply:

```
1 wf.clear_cache(lambda f: f.endswith('.zip'))
```

## Storing data

`Workflow` provides two methods to store and retrieve permanent data: `store_data()` and `stored_data()`.

These data are stored in your workflow's data directory (see `datadir`).

```
1 from workflow import Workflow
2
3 wf = Workflow()
4 wf.store_data('name', data)
5 # data will be `None` if there is nothing stored under `name`
6 data = wf.stored_data('name')
```

These methods do not support the data expiry features of the cached data methods, but you can specify your own serializer for each datastore, making it simple to store data in, e.g., JSON or YAML format.

You should use these methods (and not the data caching ones) if the data you are saving should not be deleted as part of system maintenance.

If you want to specify your own file format/serializer, please see [Serialization of stored/cached data](#) for details.

## Clearing stored data

As with cached data, there is a convenience method for deleting all the files stored in your workflow's `datadir`.

By default, `clear_data()` will delete all the files stored in `datadir`. It is used by the `workflow:deldata` and `workflow:reset` [magic arguments](#).

It is possible to selectively delete files contained in the data directory by supplying the optional `filter_func` callable. Please see [Clearing cached data](#) for details on how `filter_func` works.

## Settings

`Workflow.settings` is a subclass of `dict` that automatically saves its contents to the `settings.json` file in your Workflow's data directory when it is changed.

Settings can be used just like a normal `dict` with the caveat that all keys and values must be serializable to JSON.

**Warning:** A Settings instance can only automatically recognise when you directly alter the values of its own keys:

```
1 wf = Workflow()
2 wf.settings['key'] = {'key2': 'value'} # will be automatically saved
3 wf.settings['key']['key2'] = 'value2' # will *not* be automatically saved
```

If you've altered a data structure stored within your workflow's `Workflow.settings`, you need to explicitly call `Workflow.settings.save()`.

If you need to store arbitrary data, you can use the [cached data API](#).

If you need to store data securely (such as passwords and API keys), `Workflow` also provides [simple access to the OS X Keychain](#).

## Keychain access

Methods `Workflow.save_password(account, password)`, `Workflow.get_password(account)` and `Workflow.delete_password(account)` allow access to the Keychain. They may raise `PasswordNotFound` if no password is set for the given account or `KeychainError` if there is a problem accessing the Keychain. Passwords are stored in the user's default Keychain. By default, the Workflow's Bundle ID will be used as the service name, but this can be overridden by passing the `service` argument to the above methods.

Example usage:

```
1 from workflow import Workflow
2
3 wf = Workflow()
4
5 wf.save_password('hotmail-password', 'passwordllolz')
6
7 password = wf.get_password('hotmail-password')
8
9 wf.delete_password('hotmail-password')
10
11 # raises PasswordNotFound exception
12 password = wf.get_password('hotmail-password')
```

See [the relevant part of the tutorial](#) for a full example.

### 5.1.4 Searching/filtering data

`Workflow.filter()` provides an Alfred-like search algorithm for filtering your workflow's data. By default, `Workflow.filter()` will try to match your search query via CamelCase, substring, initials and all characters, applying different weightings to the various kind of matches (see `Workflow.filter()` for a detailed description of the algorithm and match flags).

**Warning:** Check query before calling `Workflow.filter()`. query may not be empty or contain only whitespace. This will raise a `ValueError`.  
`Workflow.filter()` is not a “little sister” of a Script Filter and won't return a list of all results if query is empty. query is *not* an optional argument and trying to filter data against a meaningless query is treated as an error.  
`Workflow.filter()` won't complain if items is an empty list, but it *will* raise a `ValueError` if query is empty.

Best practice is to do the following:

```
1 def main(wf):
2
3     query = None # Ensure `query` is initialised
4
5     # Set `query` if a value was passed (it may be an empty string)
6     if len(wf.args):
7         query = wf.args[0]
8
9     items = load_my_items_from_somewhere() # Load data from blah
10
11     if query: # Only call `filter()` if there's a `query`
12         items = wf.filter(query, items)
13
14     # Show error if there are no results. Otherwise, Alfred will show
15     # its fallback searches (i.e. "Search Google for 'XYZ'")
16     if not items:
17         wf.add_item('No items', icon=ICON_WARNING)
18
19     # Generate list of results. If `items` is an empty list,
20     # nothing will happen
21     for item in items:
22         wf.add_item(item['title'], ...)
23
24     wf.send_feedback() # Send results to Alfred via STDOUT
```

This is by no means essential (`wf.args[0]` will always be set if the script is called from Alfred via `python thescript.py "{query}"`), but it *won't* work from the command line unless called with an empty string (`python thescript.py ""`), and it's good to be aware of when you're dealing with unset/empty variables.

---

**Note:** By default, `Workflow.filter()` will match and return anything that contains all the characters in query in the same order, regardless of case. Not only can this lead to unacceptable performance when working with thousands of items, but it's also very likely that you'll want to set the standard a little higher.

See [Restricting results](#) for info on how to do that.

---

To use `Workflow.filter()`, pass it a query, a list of items to filter and sort, and if your list contains items other than strings, a key function that generates a string search key for each item:

```
1 from workflow import Workflow
2
3 names = ['Bob Smith', 'Carrie Jones', 'Harry Johnson', 'Sam Butterkeks']
4
5 wf = Workflow()
6
```

```
7 hits = wf.filter('bs', names)
```

Which returns:

```
['Bob Smith', 'Sam Butterkeks']
```

(bs are Bob Smith's initials and Butterkeks contains both letters in that order.)

If your data are not strings:

```
1 from workflow import Workflow
2
3 books = [
4     {'title': 'A damn fine afternoon', 'author': 'Bob Smith'},
5     {'title': 'My splendid adventure', 'author': 'Carrie Jones'},
6     {'title': 'Bollards and other street treasures', 'author': 'Harry Johnson'},
7     {'title': 'The horrors of Tuesdays', 'author': 'Sam Butterkeks'}
8 ]
9
10
11 def key_for_book(book):
12     return '{} {}'.format(book['title'], book['author'])
13
14 wf = Workflow()
15
16 hits = wf.filter('bot', books, key_for_book)
```

Which returns:

```
[{'author': 'Harry Johnson', 'title': 'Bollards and other street treasures'},
 {'author': 'Bob Smith', 'title': 'A damn fine afternoon'}]
```

## Restricting results

Chances are, you would not want bot to match Bob Smith A damn fine afternoon at all, or indeed any of the other books. Indeed, they have very low scores:

```
hits = wf.filter('bot', books, key_for_book, include_score=True)
```

produces:

```
[({'author': 'Bob Smith', 'title': 'A damn fine afternoon'},
  11.11111111111111,
  64),
 ({'author': 'Harry Johnson', 'title': 'Bollards and other street treasures'},
  3.333333333333335,
  64),
 ({'author': 'Sam Butterkeks', 'title': 'The horrors of Tuesdays'}, 3.125, 64)]
```

(64 is the rule that matched, MATCH\_ALLCHARS, which matches if all the characters in query appear in order in the search key, regardless of case).

---

**Tip:** rules in `filter()` results are returned as integers. To see the name of the corresponding rule, see [Matching rules](#).

---

If we filter {'author': 'Brienne of Tarth', 'title': 'How to beat up men'} and {'author': 'Zoltar', 'title': 'Battle of the Planets'}, which we probably would want to match bot, we get:

```
[({'author': 'Zoltar', 'title': 'Battle of the Planets'}, 98.0, 8),
 ({'author': 'Brienne of Tarth', 'title': 'How to beat up men'}, 90.0, 16)]
```

(The ranking would be reversed if `key_for_book()` returned `author title` instead of `title author`.)  
So in all likelihood, you'll want to pass a `min_score` argument to `Workflow.filter()`:

```
hits = wf.filter('bot', books, key_for_book, min_score=20)
```

and/or exclude some of the matching rules:

```
1 from workflow import Workflow, MATCH_ALL, MATCH_ALLCHARS
2
3 # [...]
4
5 hits = wf.filter('bot', books, key_for_book, match_on=MATCH_ALL ^ MATCH_ALLCHARS)
```

You can set match rules using bitwise operators, so `|` to combine them or `^` to remove them from `MATCH_ALL`:

```
1 # match only CamelCase and initials
2 match_on=MATCH_CAPITALS | MATCH_INITIALS
3
4 # match everything but all-characters-in-item and substring
5 match_on=MATCH_ALL ^ MATCH_ALLCHARS ^ MATCH_SUBSTRING
```

**Warning:** `MATCH_ALLCHARS` is particularly slow and provides the worst matches. You should consider excluding it, especially if you're calling `Workflow.filter()` with more than a few hundred items or expect multi-word queries.

## Diacritic folding

By default, `Workflow.filter()` will fold non-ASCII characters to approximate ASCII equivalents (e.g. `é > e`, `ü > u`) if query contains only ASCII characters. This behaviour can be turned off by passing `fold_diacritics=False` to `Workflow.filter()`.

---

**Note:** To keep the library small, only a subset of European languages are supported. The `Unidecode` library should be used for comprehensive support of non-European alphabets.

---

Users may override a Workflow's default settings via `workflow:folding...` *magic arguments*.

## “Smart” punctuation

The default diacritic folding only alters letters, not punctuation. If your workflow also works with text that contains so-called “smart” (i.e. curly) quotes or n- and m-dashes, you can use the `Workflow.dumbify_punctuation()` method to replace smart quotes and dashes with normal quotes and hyphens respectively.

## Matching rules

Here are the `MATCH_*` constants from `workflow` and their numeric values.

For a detailed description of the rules see `Workflow.filter()`.

| Name                      | Value |
|---------------------------|-------|
| MATCH_STARTSWITH          | 1     |
| MATCH_CAPITALS            | 2     |
| MATCH_ATOM                | 4     |
| MATCH_INITIALS_STARTSWITH | 8     |
| MATCH_INITIALS_CONTAIN    | 16    |
| MATCH_INITIALS            | 24    |
| MATCH_SUBSTRING           | 32    |
| MATCH_ALLCHARS            | 64    |
| MATCH_ALL                 | 127   |

### 5.1.5 Retrieving data from the web

The `unit tests` in the source repository contain examples of pretty much everything `workflow.web` can do:

- GET and POST variables
- Retrieve and decode JSON
- Post JSON
- Post forms
- Automatically handle encoding for HTML and XML
- Basic authentication
- File uploads with forms and without forms
- Download large files
- Variable timeouts
- Ignore redirects

See the [API documentation](#) for more information.

### 5.1.6 Background processes

Many workflows provide a convenient interface to applications and/or web services.

For performance reasons, it's common for workflows to cache data locally, but updating this cache typically takes a few seconds, making your workflow unresponsive while an update is occurring, which is very un-Alfred-like.

To avoid such delays, Alfred-Workflow provides the `background` module to allow you to easily run scripts in the background.

There are two functions, `run_in_background()` and `is_running()`, that provide the interface. The processes started are full daemon processes, so you can start real servers as easily as simple scripts.

Here's an example of a common usage pattern (updating cached data in the background). What we're doing is:

1. Checking the age of the cached data and running the update script via `run_in_background()` if the cached data are too old or don't exist.
2. (Optionally) informing the user that data are being updated.
3. Loading the cached data regardless of age.
4. Displaying the cached data (if any).

```

1 from workflow import Workflow, ICON_INFO
2 from workflow.background import run_in_background, is_running
3
4 def main(wf):
5     # Is cache over 1 hour old or non-existent?
```

```
6  if not wf.cached_data_fresh('exchange-rates', 3600):
7      run_in_background('update',
8                          ['/usr/bin/python',
9                          wf.workflowfile('update_exchange_rates.py')])
10
11  # Add a notification if the script is running
12  if is_running('update'):
13      wf.add_item('Updating exchange rates...', icon=ICON_INFO)
14
15  # max_age=0 will load any cached data regardless of age
16  exchange_rates = wf.cached_data('exchange-rates', max_age=0)
17
18  # Display (possibly stale) cache data
19  if exchange_rates:
20      for rate in exchange_rates:
21          wf.add_item(rate)
22
23  # Send results to Alfred
24  wf.send_feedback()
25
26  if __name__ == '__main__':
27      wf = Workflow()
28      wf.run(main)
```

For a working example, see [Part 2 of the Tutorial](#) or the [source code](#) of my [Git Repos](#) workflow, which is a bit smarter about showing the user update information.

## 5.1.7 Self-updating

New in version 1.9.

Add self-updating capabilities to your workflow. It regularly (every day by default) fetches the latest releases from the specified GitHub repository and then asks the user if they want to replace the workflow if a newer version is available.

Users may turn off automatic checks for updates using the `workflow:noautoupdate` [magic argument](#).

**Danger:** If you are not careful, you might accidentally overwrite a local version of the workflow you're working on and lose all your changes!

If you're working on a workflow, it's a good idea to make sure you increase the version number *before* you start making any changes.

See [Version numbers](#) for precise information on how Alfred-Workflow determines whether a workflow has been updated.

Currently, only updates from [GitHub releases](#) are supported.

For your workflow to be able to recognise and download newer versions, the `version` value you pass to `Workflow` **should** be one of the versions (i.e. tags) in the corresponding GitHub repo's releases list. See [Version numbers](#) for more information.

There must be one (and only one) `.alfredworkflow` binary attached to a release otherwise it will be ignored. This is the file that will be downloaded and installed via Alfred's default installation mechanism.

---

**Important:** Releases marked as pre-release on GitHub will also be ignored.

---

To use this feature, you must pass a `dict` as the `update_settings` argument to `Workflow`. It **must** have the key/value pair `github_slug`, which is your username and the name of the workflow's repo in the format `username/reponame`. The version of the currently installed workflow must also be specified. You can do this in the `update_settings` dict or in a `version` file in the root of your workflow (next to `info.plist`), e.g.:



```

1 from workflow import Workflow
2
3 __version__ = '1.1'
4
5 ...
6
7 wf = Workflow(..., update_settings={
8     # Your username and the workflow's repo's name
9     'github_slug': 'username/reponame',
10    # The version (i.e. release/tag) of the installed workflow
11    # If a `version` file exists in the root of your workflow,
12    # this key may be omitted
13    'version': __version__,
14    # Optional number of days between checks for updates
15    'frequency': 7
16 }, ...)
17
18 ...
19
20 if wf.update_available:
21     wf.start_update()

```

Or alternatively, create a version file in the root directory or your workflow alongside `info.plist`:

```

Your Workflow/
  icon.png
  info.plist
  yourscript.py
  version
  workflow/
    ...
    ...

```

The version file should be plain text with no file extension and contain nothing but the version string, e.g.:

```
1.2.5
```

Using a version file:

```

1 from workflow import Workflow
2
3 ...
4
5 wf = Workflow(..., update_settings={
6     # Your username and the workflow's repo's name
7     'github_slug': 'username/reponame',
8     # Optional number of days between checks for updates
9     'frequency': 7
10 }, ...)
11
12 ...
13
14 if wf.update_available:
15     wf.start_update()

```

Please see [Versioning and migration](#) for detailed information on the required version number format and associated features.

**Note:** Alfred-Workflow will automatically check in the background if a newer version of your workflow is available, but will *not* automatically inform the user nor download and install the update.

To view update status/install a newer version, the user must either call one of your workflow's Script Filters with the workflow:update *magic argument*, in which case Alfred-Workflow will handle the up-

date automatically, or you must add your own update action using `Workflow.update_available` and `Workflow.start_update()` to check for and install newer versions respectively.

The `check_update()` method is called automatically when you create a `workflow.workflow.Workflow` object. If sufficient time has elapsed since the last check (1 day by default), it starts a background process that checks for new releases. You can alter the update interval with the optional frequency key in `update_settings` dict (see the *example above*).

`Workflow.update_available` is `True` if an update is available, and `False` otherwise.

`Workflow.start_update()` returns `False` if no update is available, or if one is, it will return `True`, download the newer version and tell Alfred to install it.

If you want more control over the update mechanism, you can use `update.check_update()` directly. It caches information on the latest available release under the cache key `__workflow_update_status`, which you can access via `Workflow.cached_data()`.

Users can turn off automatic checks for updates with the `workflow:noautoupdate` *magic argument* and back on again with `workflow:autoupdate`.

## Version numbers

Please see *Versioning and migration* for detailed information on the required version number format and associated features.

### 5.1.8 Versioning and migration

New in version 1.10.

If you intend to distribute your workflow, it's a good idea to use version numbers. It allows users to see if they're using an out-of-date version, and more importantly, it allows you to know which version a user has when they ask you for support or to fix a bug (that you may already have fixed).

If your workflow has a version number set (see *Setting a version number*), the version will be logged every time the workflow is run to help with debugging, and can also be displayed using the `workflow:version` *magic argument*.

If you wish to use the *self-updating feature*, your workflow must have a version number.

Having a version number also enables the first run/migration functionality. See *First run/migration* below for details.

#### Setting a version number

There are two ways to set a version number. The simplest and best is to create a `version` file in the root directory of your workflow (next to `info.plist`) that contains the version number:

```
Your Workflow/
    icon.png
    info.plist
    yoursript.py
    version
    workflow/
    ...
```

You may also specify the version number using the `version` key in the `update_settings` dictionary passed to `Workflow`, though you can only use this method if your workflow supports self-updates from GitHub.

Using a `version` file is preferable as then you only need to maintain the version number in one place.

## Version numbers

In version 1.10 and above, Alfred-Workflow requires *Semantic versioning*, which is the format GitHub also expects. Alfred-Workflow deviates from the semantic versioning standard slightly, most notably in that you don't have to specify a minor or patch version, i.e. 1.0 is fine, as is simply 1 (the standard requires these to both be written 1.0.0). See *Semantic versioning* for more details on version formatting.

The *de-facto* way to tag releases on GitHub is use a semantic version number preceded by v, e.g. v1.0, v2.3.1 etc., whereas the *de-facto* way to version Python libraries is to do the same, but without the preceding v, e.g. 1.0, 2.3.1 etc.

As a result, Alfred-Workflow will strip a preceding v from both local and remote versions (i.e. you can specify 1.0 or v1.0 in either or both of your Python code and GitHub releases).

When this is done, if the latest GitHub version is higher than the local version, Alfred-Workflow will consider the remote version to be an update.

Thus, calling `Workflow` with `update_settings={'version': '1.2', ...}` or `update_settings={'version': 'v1.2', ...}` will be considered the same version as the GitHub release tag v1.2 or 1.2 (or indeed 1.2.0).

## Semantic versioning

Semantic versioning is a standard for formatting software version numbers.

Essentially, a version number must consist of a major version number, a minor version number and a patch version number separated by dots, e.g. 1.0.1, 2.10.3 etc. You should increase the patch version when you fix bugs, the minor version when you add new features and the major version if you change the API.

You may also add additional pre-release version info to the end of the version number, preceded by a hyphen (-), e.g. 2.0.0-rc.1 or 2.0.0-beta.

Alfred-Workflow differs from the standard in that you aren't required to specify a minor or patch version, i.e. 1.0 is fine, as is 1 (and both are considered equal and also equal to 1.0.0).

This change was made as relatively few workflow authors use patch versions.

See the [semantic versioning](#) website for full details of the standard and the rationale behind it.

## First run/migration

New in version 1.10.

If your workflow uses *version numbers*, you can use the `Workflow.first_run` and `Workflow.last_version_run` attributes to bootstrap newly-installed workflows or to migrate data from an older version.

`first_run` will be True if this version of the workflow has never run before. If an older version has previously run, `last_version_run` will contain the version of that workflow.

Both `last_version_run` and `version` are `Version` instances (or None) to make comparison easy. Be sure to check for None before comparing them: comparing `Version` and None will raise a `ValueError`.

`last_version_run` is set to the value of the currently running workflow if it runs successfully without raising an exception.

---

**Important:** `last_version_run` will only be set automatically if you run your workflow via `Workflow.run()`. This is because `Workflow` is often used as a utility class by other workflow scripts, and you don't want your background update script to confuse things by setting the wrong version.

If you want to set `last_version_run` yourself, use `set_last_version()`.

---






















### 5.1.9 System icons

The *workflow* module provides access to a number of default OS X icons via `ICON_*` constants for use when generating Alfred feedback:

```
1 from workflow import Workflow, ICON_INFO
2
3 wf = Workflow()
4 wf.add_item('For your information', icon=ICON_INFO)
5 wf.send_feedback()
```

#### List of icons

These are all the icons accessible in *workflow*. They (and more) can be found in `/System/Library/CoreServices/CoreTypes.bundle/Contents/Resources/`.

| Name             | Preview   |
|------------------|---|
| ICON_ACCOUNT     |    |
| ICON_BURN        |    |
| ICON_CLOCK       |    |
| ICON_COLOR       |    |
| ICON_COLOUR      |    |
| ICON_EJECT       |    |
| ICON_ERROR       |    |
| ICON_FAVORITE    |    |
| ICON_FAVOURITE   |   |
| ICON_GROUP       |  |
| ICON_HELP        |  |
| ICON_HOME        |  |
| ICON_INFO        |  |
| ICON_NETWORK     |  |
| ICON_NOTE        |  |
| ICON_SETTINGS    |  |
| ICON_SWIRL       |  |
| ICON_SWITCH      |  |
| ICON_SYNC        |  |
|                  |  |
| 5.1. User Manual |   |
| ICON_USER        |  |

If you'd like other standard OS X icons to be added, please [add an issue on GitHub](#).

### 5.1.10 “Magic” arguments

If your Script Filter (or script) accepts a query (or command line arguments), you can pass it so-called magic arguments that instruct *Workflow* to perform certain actions, such as opening the log file or clearing the cache/settings.

These can be a big help while developing and debugging and especially when debugging problems your Workflow's users may be having.

The `Workflow.run()` method (which you should “wrap” your Workflow's entry functions in) will catch any raised exceptions, log them and display them in Alfred. You can call your Workflow with `workflow:openlog` as an Alfred query/command line argument and *Workflow* will open the Workflow's log file in the default app (usually **Console.app**).

This makes it easy for you to get at the log file and data and cache directories (hidden away in `~/Library`), and for your users to send you their logs for debugging.

---

**Note:** Magic arguments will only work with scripts that accept arguments *and* use the `args` property (where magic arguments are parsed).

---

*Workflow* supports the following magic arguments by default:

- `workflow:magic` — List available magic arguments.
- `workflow:help` — Open workflow's help URL in default web browser. This URL is specified in the `help_url` argument to *Workflow*.
- `workflow:version` — Display the installed version of the workflow (if one is set).
- `workflow:delcache` — Delete the Workflow's cache.
- `workflow:deldata` — Delete the Workflow's saved data.
- `workflow:delsettings` — Delete the Workflow's settings file (which contains the data stored using *Workflow.settings*).
- `workflow:foldingdefault` — Reset diacritic folding to workflow default
- `workflow:foldingoff` — Never fold diacritics in search keys
- `workflow:foldingon` — Force diacritic folding in search keys (e.g. convert *ü* to *ue*)
- `workflow:opencache` — Open the Workflow's cache directory.
- `workflow:opendata` — Open the Workflow's data directory.
- `workflow:openlog` — Open the Workflow's log file in the default app.
- `workflow:openterm` — Open a Terminal window in the Workflow's root directory.
- `workflow:openworkflow` — Open the Workflow's root directory (where `info.plist` is).
- `workflow:reset` — Delete the Workflow's settings, cache and saved data.
- `workflow:update` — Check for a newer version of the workflow using GitHub releases and install the newer version if one is available.
- `workflow:noautoupdate` — Turn off automatic checks for updates.
- `workflow:autoupdate` — Turn automatic checks for updates on.

The three `workflow:folding...` settings allow users to override the diacritic folding set by a workflow's author. This may be useful if the author's choice does not correspond with a user's usage pattern.

You can turn off magic arguments by passing `capture_args=False` to *Workflow* on instantiation, or call the corresponding methods of *Workflow* directly, perhaps assigning your own keywords within your Workflow:

- `open_help()`
- `open_log()`
- `open_cachedir()`
- `open_datadir()`
- `open_workflowdir()`
- `open_terminal()`
- `clear_cache()`
- `clear_data()`
- `clear_settings()`
- `reset()` (a shortcut to call the three previous `clear_*` methods)
- `check_update()`
- `start_update()`

### Customising magic arguments

The default prefix for magic arguments (`workflow:`) is contained in the `magic_prefix` attribute of `Workflow`. If you want to change it to, say, `wf:` (which will become the default in v2 of Alfred-Workflow), simply reassign it:

```
wf.magic_prefix = 'wf:'
```

The magic arguments are defined in the `Workflow.magic_arguments` dictionary. The dictionary keys are the keywords for the arguments (without the prefix) and the values are functions that should be called when the magic argument is entered. You can show a message in Alfred by returning a unicode string from the function.

To add a new magic argument that opens the workflow's settings file, you could do:

```
1 wf = Workflow()
2 wf.magic_prefix = 'wf:' # Change prefix to `wf:`
3
4 def opensettings():
5     subprocess.call(['open', wf.settings_path])
6     return 'Opening workflow settings...'
7
8 wf.magic_arguments['settings'] = opensettings
```

Now entering `wf:settings` as your workflow's query in Alfred will open `settings.json` in the default application.

### 5.1.11 Serialization of stored/cached data

By default, both cache and data files (created using the APIs described in *Persistent data*) are cached using `cPickle`. This provides a great compromise in terms of speed and the ability to store arbitrary objects.

When changing or specifying a serializer, use the name under which the serializer is registered with the `workflow.manager` object.

**Warning:** When it comes to cache data, it is *strongly recommended* to stick with the default. `cPickle` is very fast and fully supports standard Python data structures (dict, list, tuple, set etc.). If you really must customise the cache data format, you can change the default cache serialization format to `pickle` thus:

```
1 wf = Workflow()
2 wf.cache_serializer = 'pickle'
```

Unlike the stored data API, the cached data API can't determine the format of the cached data. If you change the serializer without clearing the cache, errors will probably result as the serializer tries to load data in a foreign format.

In the case of stored data, you are free to specify either a global default serializer or one for each individual datastore:

```
1 wf = Workflow()
2 # Use `pickle` as the global default serializer
3 wf.data_serializer = 'pickle'
4
5 # Use the JSON serializer only for these data
6 wf.store_data('name', data, serializer='json')
```

This is primarily so you can create files that are human-readable or useable by other software. The generated JSON is formatted to make it readable.

The `stored_data()` method can automatically determine the serialization of the stored data (based on the file extension, which is the same as the name the serializer is registered under), provided the corresponding serializer is registered. If it isn't, a `ValueError` will be raised.

## Built-in serializers

There are 3 built-in, pre-configured serializers:

- `cpickle` — the default serializer for both cached and stored data, with very good support for native Python data types;
- `pickle` — a more flexible, but much slower alternative to `cpickle`; and
- `json` — a very common data format, but with limited support for native Python data types.

See the built-in `cPickle`, `pickle` and `json` libraries for more information on the serialization formats.

## Managing serializers

You can add your own serializer, or replace the built-in ones, using the configured instance of `SerializerManager` at `workflow.manager`, e.g. `from workflow import manager`.

A serializer object must have `load()` and `dump()` methods that work the same way as in the built-in `json` and `pickle` libraries, i.e.:

```
1 # Reading
2 obj = serializer.load(open('filename', 'rb'))
3 # Writing
4 serializer.dump(obj, open('filename', 'wb'))
```

To register a new serializer, call the `register()` method of the `workflow.manager` object with the name of the serializer and the object that performs serialization:

```
1 from workflow import Workflow, manager
2
3
4 class MySerializer(object):
```



```

5
6     @classmethod
7     def load(cls, file_obj):
8         # load data from file_obj
9
10    @classmethod
11    def dump(cls, obj, file_obj):
12        # serialize obj to file_obj
13
14    manager.register('myformat', MySerializer())

```

---

**Note:** The name you specify for your serializer will be the file extension of the stored files.

---

## Serializer interface

A serializer **must** conform to this interface (like `json` and `pickle`):

```

1 serializer.load(file_obj)
2 serializer.dump(obj, file_obj)

```

See the [Serialization](#) section of the API documentation for more information.

## 5.1.12 Encoded strings and Unicode

This is a brief guide to Unicode and encoded strings aimed at Alfred-Workflow users (and Python coders in general) who are unfamiliar with them.

Encoding errors are *by far* the most common group of bugs in Python workflows in the wild (they're so easy for developers to miss).

This guide should give you an idea of what Unicode and encoded strings are, and why and how you as a workflow developer should deal with them.

---

**Important:** String encoding is something Python 2 will let you largely ignore. It will happily let you mix strings of different encodings without complaint (although the result will most likely be garbage) and if you mix Unicode and encoded strings, Python will silently “promote” the encoded string to Unicode by decoding it as ASCII. If your workflow only ever uses ASCII, you need never worry about Unicode or string encoding.

But make no mistake: if you distribute your workflow, somebody *will* feed your workflow non-ASCII text. Although Alfred is English-only, it's not used exclusively by monolingual English speakers. What's more, standard English-language characters, like £ or €, are also non-ASCII.

**If you intend to distribute your workflow, you should make sure it works with non-ASCII text.**

If you don't, I guarantee a text-encoding issue will be one of the first bug reports.

---

## TL;DR

Best practice in Python programs is to use Unicode internally and decode all text input and encode all text output at IO boundaries (i.e. right where it enters/leaves your program). On OS X, UTF-8 is almost always the right encoding.

Be sure to decode all input from and encode all output to the system (in particular via `subprocess` and when passing a `{query}` to a subsequent workflow action).

If you don't, your workflow *will* break or, at best, not work as intended when someone feeds it non-ASCII text.

Alfred-Workflow will almost always give you Unicode strings. (The exception is `web.Response`, whose `text()` method will return an encoded string if it couldn't determine the encoding.)

Use `Workflow.decode()` to decode input and `u'My unicode string'.encode('utf-8')` to encode output, e.g.:

```
1  #!/usr/bin/python
2  # encoding: utf-8
3
4  # Because we want to work with Unicode, it's simpler if we make
5  # literal strings in source code Unicode strings by default, so
6  # we set `encoding: utf-8` at the very top of the script and
7  # import `unicode_literals` before any code
8  #
9  # See Tip further down the page for more info
10 from __future__ import unicode_literals, print_function
11
12 import subprocess
13 from workflow import Workflow
14
15 wf = Workflow()
16 # wf.args decodes and normalizes sys.argv for you
17 query = wf.args[0]
18 # `subprocess` returns encoded strings (UTF-8 in this case)
19 # Note: the arguments are prefixed with `b` because of unicode_literals
20 # You should pass encoded strings to `subprocess`. It doesn't much
21 # matter in this case, as everything can be encoded to ASCII, but if you're
22 # passing in, say, a user-supplied query, be sure to encode it to UTF-8
23 output = subprocess.check_output([b'mdfind', b'-onlyin',
24                                   os.getenv('HOME'),
25                                   b'kind:folder date:today'])
26 # Convert to Unicode and NFC-normalize
27 output = wf.decode(output)
28 # Split the output into individual filepaths
29 paths = [s.strip() for s in output.split('\n') if s.strip()]
30 # Filter paths by query
31 paths = wf.filter(query, paths,
32                   # We just want to filter on filenames, not the whole path
33                   key=lambda s: os.path.basename(s),
34                   min_score=30)
35
36 if paths:
37     # For demonstration purposes, pass the first result as `{query}`
38     # to the next workflow Action.
39     print(paths[0].encode('utf-8'))
```

## String types

In Python, there are two different kind of strings: Unicode and encoded strings.

Unicode strings only exist within running programs (Unicode is a concept rather than a concrete implementation), while encoded strings are binary data that are encoded according to some scheme that maps characters to a specific binary representation (e.g. UTF-8 or ASCII).

In Python, these have the types `unicode` and `str` respectively.

As noted, Unicode strings only exist within a running program. Any text stored on disk, passed into or out of a program or transmitted over a network *must* be encoded. On OS X, almost all text (e.g. filenames, most text output from programs) is encoded with UTF-8.

In order for your program to work properly, it's important to ensure that all text is of the same type/encoding:

```
>>> u = u'Fahrvergnügen' # This is a Unicode string
>>> enc1 = u.encode('utf-8') # OS X default encoding
>>> enc2 = u.encode('latin-1') # Older standard German encoding
>>> enc1 == enc2
```

```
False
>>> u == enc1
UnicodeWarning: Unicode equal comparison failed to convert both arguments to Unicode - interpreting
False
>>> unicode(enc1, 'utf-8') == unicode(enc2, 'latin-1')
True
```

The correct way to do this in Python is to decode all text input to Unicode as soon as it enters your program. In particular, this means:

- Command-line arguments (via `sys.argv`)
- Environmental variables (via `os.environ`)
- The contents of text files (via `open()`)
- Data retrieved from the web (via `urllib.urlopen()`)
- The output of subprocesses (via `subprocess.check_output()` or `subprocess.Popen` etc.)
- Filepaths (via `os.listdir()` etc.). Sometimes. Basically, if you pass a Unicode string to a filesystem function, you'll get Unicode back. If you pass an encoded string, you'll get an encoded (UTF-8) string back.

Alfred-Workflow uses Unicode throughout, and any command-line arguments (`Workflow.args`), environmental variables (`Workflow.alfred_env`), or data from the web (e.g. `web.Response.text`) will be decoded to Unicode for you.

As a result of this, it's important that you also decode any text your workflow pulls in from other sources. When you combine Unicode and encoded strings in Python 2, Python will “promote” the encoded string to Unicode by attempting to decode it as ASCII. In many cases this will work, but if the encoded string contains characters that aren't in ASCII (e.g. £ or ü or —), your workflow will die in flames.

**Tip:** Always test your workflow with non-ASCII input to flush out any accidental mixing of Unicode and encoded strings.

`Workflow` provides the convenience method `Workflow.decode()` for working with Unicode and encoded strings. You can pass it Unicode or encoded strings and it will return normalized Unicode. You can specify the encoding and normalization form with the `input_encoding` and `normalization` arguments to `Workflow` or with the `encoding` and `normalization` arguments to `Workflow.decode()`. Generally, you shouldn't need to change the default encoding of UTF-8, which is what OS X uses, but you may need to alter the normalization depending on where your workflow gets its data from.

**Tip:** To save yourself from having to prefix every string in your source code with `u` to mark it as a Unicode string, add `from __future__ import unicode_literals` at the top of your Python scripts. This makes all unprefix strings Unicode by default (use `b' '` to create an encoded string):

```
1 # encoding: utf-8
2 from __future__ import unicode_literals
3
4 ustr = 'This is a Unicode string'
5 bstr = b'This is an encoded string'
```

## Normalization

Unicode provides multiple ways to represent the same character. Normalization is the process of ensuring that all instances of a given Unicode character are represented in the same way.

### TL;DR

Normalize *all* input.

If your workflow is based around comparing a user query to data from the system (filepaths, output of command-line programs), you should instantiate `Workflow` with the `normalization='NFD'` argument.

If your workflow uses data from the Web (via native Python libraries, including `web`), you probably don't need to do anything (everything will be NFC-normalized).

If you're mixing both kinds of data, the simplest solution is probably to run all data from the system through `Workflow.decode()` to ensure it is normalized in the same way as data from the Web.

### Why does normalization matter?

In Unicode, accented characters can be represented in different ways, e.g. `ü` can be represented as `ü` or as `u+''`. Unfortunately, Python isn't smart enough to ensure that all Unicode strings are normalized to use the same representations when comparing them.

Therefore, if you're comparing a string containing `ü` entered by the user in Alfred's query box or in the source code (which will be NFC-normalized by default when using Alfred-Workflow) with an ostensibly identical string that came from OS X's filesystem (which is NFD-normalized), Python won't recognise them as being the same:

```

1 >>> from unicodedata import normalize
2 >>> from glob import glob
3 >>> name = u'München.txt' # German for 'Munich'. NFC-normalized, as it's Python source code
4 >>> print(repr(name))
5 u'M\xfcncchen.txt'
6 >>> open(name, 'wb').write('') # Create an empty text file called `München.txt`
7
8 >>> for filename in glob(u'*.txt'):
9 ...     if filename == name:
10 ...         print(u'Match : {0} ({0!r}) == {1} ({1!r})'.format(filename, name))
11 ...     else:
12 ...         print(u'No match : {0} ({0!r}) != {1} ({1!r})'.format(filename, name))
13 ...
14 # The filename has been NFD-normalized by the filesystem
15 No match : München.txt (u'Mu\u0308ncchen.txt') != München.txt (u'M\xfcncchen.txt')
16 >>> for filename in glob(u'*.txt'):
17 ...     filename = normalize('NFC', filename) # Ensure the same normalization
18 ...     if filename == name:
19 ...         print(u'Match : {0} ({0!r}) == {1} ({1!r})'.format(filename, name))
20 ...     else:
21 ...         print(u'No match : {0} ({0!r}) != {1} ({1!r})'.format(filename, name))
22 ...
23 Match : München.txt (u'M\xfcncchen.txt') == München.txt (u'M\xfcncchen.txt')

```

As a result of this Python quirk (Python 3 is alas no better in this regard), it's important to ensure that all input is normalized in the same way or, for example, a user-provided query (NFC-normalized by default) may not match the output of a shell command run via `subprocess` (NFD-normalized) even though they are ostensibly the same.

### Normalization with Alfred-Workflow

By default, `Workflow` and `web` return command line arguments from Alfred and text/decoded JSON data respectively as NFC-normalized Unicode strings.

This is the default for Python. You can change this via the `normalization` keyword to `Workflow` (this will, however, not affect `web`, which *always* returns NFC-encoded Unicode strings).

If your workflow works with data from the system (via `subprocess`, `os.listdir()` etc.), you should almost certainly be NFC-normalizing those strings or changing the default normalization to **NFD**, which is (more or less) what OS X uses. `Workflow.decode()` can help with this.

If you pass a Unicode string to `Workflow.decode()`, it will be normalized using the form passed in the `normalization` argument to `Workflow.decode()` or to `Workflow` on instantiation.

If you pass an encoded string, it will be decoded to Unicode with the encoding passed in the `encoding` argument to `Workflow.decode()` or the `input_encoding` argument to `Workflow` on instantiation and then normalized as above.

### Further information

If you're unfamiliar with using Unicode in Python, have a look at the official Python [Unicode HOWTO](#).



---

## API documentation

---

Documetation of the Alfred-Workflow APIs generated from the source code. A handy reference if (like me) you sometimes forget parameter names.

### 6.1 Alfred-Workflow API

This API documentation describes how Alfred-Workflow is put together.

See *User Manual* for documentation focussed on performing specific tasks.

#### 6.1.1 The Workflow Object

The *Workflow* object is the main interface to this library.

See *Workflow setup and skeleton* in the *User Manual* for an example of how to set up your Python script to best utilise the *Workflow* object.

```
class workflow.workflow.Workflow (default_settings=None,          update_settings=None,
                                   input_encoding=u'utf-8',    normalization=u'NFC',  cap-
                                   ture_args=True, libraries=None, help_url=None)
```

Create new *Workflow* instance.

##### Parameters

- **default\_settings** (*dict*) – default workflow settings. If no settings file exists, *Workflow.settings* will be pre-populated with *default\_settings*.
- **update\_settings** (*dict*) – settings for updating your workflow from GitHub. This must be a *dict* that contains *github\_slug* and *version* keys. *github\_slug* is of the form *username/repo* and *version* **must** correspond to the tag of a release. See *Self-Updating* for more information.
- **input\_encoding** (*unicode*) – encoding of command line arguments
- **normalization** (*unicode*) – normalisation to apply to CLI args. See *Workflow.decode()* for more details.
- **capture\_args** (*Boolean*) – capture and act on *workflow:\** arguments. See *Magic arguments* for details.
- **libraries** (*tuple* or *list*) – sequence of paths to directories containing libraries. These paths will be prepended to *sys.path*.
- **help\_url** (*unicode* or *str*) – URL to webpage where a user can ask for help with the workflow, report bugs, etc. This could be the GitHub repo or a page on AlfredForum.com. If your workflow throws an error, this URL will be displayed in the log and Alfred's debugger. It can also be opened directly in a web browser with the *workflow:help magic argument*.

**add\_item**(title, subtitle=u'', modifier\_subtitles=None, arg=None, autocomplete=None, valid=False, uid=None, icon=None, icontype=None, type=None,argetext=None, copytext=None)

Add an item to be output to Alfred

Passes arguments through to XMLGenerator method `add_item()`.

#### **alfred\_env**

Alfred's environmental variables minus the `alfred_` prefix.

New in version 1.7.

The variables Alfred 2.4+ exports are:

| Variable                                  | Description  |
|---|--|
| <code>alfred_preferences</code>           | Path to Alfred.alfredpreferences (where your workflows and settings are stored).   |
| <code>alfred_preferences_localhash</code> | Machine-specific preferences are stored in <code>Alfred.alfredpreferences/preferences/local/&lt;hash&gt;</code> (see <code>alfred_preferences</code> above for the path to Alfred.alfredpreferences) |
| <code>alfred_theme</code>                 | ID of selected theme   |
| <code>alfred_theme_background</code>      | Background colour of selected theme in format <code>rgba(r, g, b, a)</code>  |
| <code>alfred_theme_subtext</code>         | Show result subtext. 0 = Always, 1 = Alternative actions only, 2 = Selected result only, 3 = Never   |
| <code>alfred_version</code>               | Alfred version number, e.g. '2.4'  |
| <code>alfred_version_build</code>         | Alfred build number, e.g. 277  |
| <code>alfred_workflow_bundleid</code>     | Bundle ID, e.g. <code>net.deanishe.alfred-mailto</code>  |
| <code>alfred_workflow_cache</code>        | Path to workflow's cache directory   |
| <code>alfred_workflow_data</code>         | Path to workflow's data directory  |
| <code>alfred_workflow_name</code>         | Name of current workflow   |
| <code>alfred_workflow_uid</code>          | UID of workflow  |

**Note:** all values are Unicode strings except `version_build` and `theme_subtext`, which are integers.

**Returns** dict of Alfred's environmental variables without the `alfred_` prefix, e.g. `preferences, workflow_data`.

#### **args**

Return command line args as normalised unicode.

Args are decoded and normalised via `decode()`.

The encoding and normalisation are the `input_encoding` and `normalization` arguments passed to `Workflow` (UTF-8 and NFC are the defaults).

If `Workflow` is called with `capture_args=True` (the default), `Workflow` will look for certain `workflow:*` args and, if found, perform the corresponding actions and exit the workflow.

See *Magic arguments* for details.

#### **bundleid**

Workflow bundle ID from environmental vars or `info.plist`.

**Returns** bundle ID



**Return type** unicode

**cache\_data** (*name*, *data*)

Save data to cache under name.

If data is None, the corresponding cache file will be deleted.

**Parameters**

- **name** – name of datastore
- **data** – data to store. This may be any object supported by the cache serializer

**cache\_serializer**

Name of default cache serializer.

New in version 1.8.

This serializer is used by `cache_data()` and `cached_data()`

See `SerializerManager` for details.

**Returns** serializer name

**Return type** unicode

**cached\_data** (*name*, *data\_func=None*, *max\_age=60*)

Retrieve data from cache or re-generate and re-cache data if stale/non-existent. If *max\_age* is 0, return cached data no matter how old.

**Parameters**

- **name** – name of datastore
- **data\_func** (callable) – function to (re-)generate data.
- **max\_age** (int) – maximum age of cached data in seconds

**Returns** cached data, return value of *data\_func* or None if *data\_func* is not set

**cached\_data\_age** (*name*)

Return age of data cached at *name* in seconds or 0 if cache doesn't exist

**Parameters** **name** (unicode) – name of datastore

**Returns** age of datastore in seconds

**Return type** int

**cached\_data\_fresh** (*name*, *max\_age*)

Is data cached at *name* less than *max\_age* old?

**Parameters**

- **name** – name of datastore
- **max\_age** (int) – maximum age of data in seconds

**Returns** True if data is less than *max\_age* old, else False

**cachedir**

Path to workflow's cache directory.

The cache directory is a subdirectory of Alfred's own cache directory in `~/Library/Caches`. The full path is:

`~/Library/Caches/com.runningwithcrayons.Alfred-2/Workflow  
Data/<bundle id>`

**Returns** full path to workflow's cache directory

**Return type** unicode

**cachefile** (*filename*)

Return full path to *filename* within your workflow's *cache directory*.

**Parameters** **filename** (unicode) – basename of file

**Returns** full path to file within cache directory

**Return type** unicode

**check\_update** (*force=False*)

Call update script if it's time to check for a new release

New in version 1.9.

The update script will be run in the background, so it won't interfere in the execution of your workflow.

See *Self-updating* in the *User Manual* for detailed information on how to enable your workflow to update itself.

**Parameters** **force** (Boolean) – Force update check

**clear\_cache** (*filter\_func=<function <lambda>>>*)

Delete all files in workflow's *cachedir*.

**Parameters** **filter\_func** (callable) – Callable to determine whether a file should be deleted or not. *filter\_func* is called with the filename of each file in the data directory. If it returns True, the file will be deleted. By default, *all* files will be deleted.

**clear\_data** (*filter\_func=<function <lambda>>>*)

Delete all files in workflow's *datadir*.

**Parameters** **filter\_func** (callable) – Callable to determine whether a file should be deleted or not. *filter\_func* is called with the filename of each file in the data directory. If it returns True, the file will be deleted. By default, *all* files will be deleted.

**clear\_settings** ()

Delete workflow's *settings\_path*.

**data\_serializer**

Name of default data serializer.

New in version 1.8.

This serializer is used by *store\_data()* and *stored\_data()*

See *SerializerManager* for details.

**Returns** serializer name

**Return type** unicode

**datadir**

Path to workflow's data directory.

The data directory is a subdirectory of Alfred's own data directory in ~/Library/Application Support. The full path is:

~/Library/Application Support/Alfred 2/Workflow Data/<bundle id>

**Returns** full path to workflow data directory

**Return type** unicode

**datafile** (*filename*)

Return full path to *filename* within your workflow's *data directory*.

**Parameters** **filename** (unicode) – basename of file

**Returns** full path to file within data directory

**Return type** unicode

**decode** (*text*, *encoding=None*, *normalization=None*)

Return text as normalised unicode.

If *encoding* and/or *normalization* is *None*, the *input\_encoding* and *normalization* parameters passed to *Workflow* are used.

#### Parameters

- **text** (encoded or Unicode string. If *text* is already a Unicode string, it will only be normalised.) – string
- **encoding** (unicode or *None*) – The text encoding to use to decode *text* to Unicode.
- **normalization** (unicode or *None*) – The normalisation form to apply to *text*.

**Returns** decoded and normalised unicode

**delete\_password** (*account*, *service=None*)

Delete the password stored at *service/account*. Raises *PasswordNotFound* if *account* is unknown.

#### Parameters

- **account** (unicode) – name of the account the password is for, e.g. “Pinboard”
- **service** (unicode) – Name of the service. By default, this is the workflow’s bundle ID

**filter** (*query*, *items*, *key=<function <lambda>>*, *ascending=False*, *include\_score=False*, *min\_score=0*, *max\_results=0*, *match\_on=127*, *fold\_diacritics=True*)

Fuzzy search filter. Returns list of *items* that match *query*.

*query* is case-insensitive. Any item that does not contain the entirety of *query* is rejected.

See *workflow.search.filter()* for detailed documentation.

**first\_run**

Return *True* if it’s the first time this version has run.

New in version 1.9.10.

Raises a *ValueError* if *version* isn’t set.

**fold\_to\_ascii** (*text*)

Convert non-ASCII characters to closest ASCII equivalent.

New in version 1.3.

---

**Note:** This only works for a subset of European languages.

---

**Parameters** **text** (unicode) – text to convert

**Returns** text containing only ASCII characters

**Return type** unicode

**get\_password** (*account*, *service=None*)

Retrieve the password saved at *service/account*. Raise *PasswordNotFound* exception if password doesn’t exist.

#### Parameters

- **account** (unicode) – name of the account the password is for, e.g. “Pinboard”
- **service** (unicode) – Name of the service. By default, this is the workflow’s bundle ID

**Returns** account password

**Return type** unicode

**info**

dict of info.plist contents.

**item\_class**

alias of Item

**last\_version\_run**

Return version of last version to run (or None)

New in version 1.9.10.

**Returns** Version instance or None

**logfile**

Return path to logfile

**Returns** path to logfile within workflow's cache directory

**Return type** unicode

**logger**

Create and return a logger that logs to both console and a log file.

Use `open_log()` to open the log file in Console.

**Returns** an initialised `Logger`

**magic\_arguments = None**

Mapping of available magic arguments. The built-in magic arguments are registered by default. To add your own magic arguments (or override built-ins), add a key:value pair where the key is what the user should enter (prefixed with `magic_prefix`) and the value is a callable that will be called when the argument is entered. If you would like to display a message in Alfred, the function should return a unicode string.

By default, the magic arguments documented [here](#) are registered.

**magic\_prefix = None**

The prefix for all magic arguments. Default is workflow:

**name**

Workflow name from Alfred's environmental vars or info.plist.

**Returns** workflow name

**Return type** unicode

**open\_cachedir()**

Open the workflow's `cachedir` in Finder.

**open\_datadir()**

Open the workflow's `datadir` in Finder.

**open\_help()**

Open `help_url` in default browser

**open\_log()**

Open workflows `logfile` in standard application (usually Console.app).

**open\_terminal()**

Open a Terminal window at workflow's `workflowdir`.

**open\_workflowdir()**

Open the workflow's `workflowdir` in Finder.

**reset()**

Delete `settings`, `cache` and `data`

**run(func)**

Call `func` to run your workflow

**Parameters** **func** – Callable to call with `self` (i.e. the *Workflow* instance) as first argument.

`func` will be called with *Workflow* instance as first argument.

`func` should be the main entry point to your workflow.

Any exceptions raised will be logged and an error message will be output to Alfred.

**save\_password** (*account*, *password*, *service=None*)

Save account credentials.

If the account exists, the old password will first be deleted (Keychain throws an error otherwise).

If something goes wrong, a `base.KeychainError` exception will be raised.

**Parameters**

- **account** (unicode) – name of the account the password is for, e.g. “Pinboard”
- **password** (unicode) – the password to secure
- **service** (unicode) – Name of the service. By default, this is the workflow’s bundle ID

**send\_feedback** ()

Print stored items to console/Alfred as XML.

**set\_last\_version** (*version=None*)

Set *last\_version\_run* to current version

New in version 1.9.10.

**Parameters** **version** (Version instance or unicode) – version to store (default is current version)

**Returns** True if version is saved, else False

**settings**

Return a dictionary subclass that saves itself when changed.

See *Settings* in the *User Manual* for more information on how to use *settings* and **important limitations** on what it can do.

**Returns** PersistentDict instance initialised from the data in JSON file at *settings\_path* or if that doesn’t exist, with the default\_settings dict passed to *Workflow* on instantiation.

**Return type** PersistentDict instance

**settings\_path**

Path to settings file within workflow’s data directory.

**Returns** path to settings.json file

**Return type** unicode

**start\_update** ()

Check for update and download and install new workflow file

New in version 1.9.

See *Self-updating* in the *User Manual* for detailed information on how to enable your workflow to update itself.

**Returns** True if an update is available and will be installed, else False

**store\_data** (*name*, *data*, *serializer=None*)

Save data to data directory.

New in version 1.8.

If *data* is None, the datastore will be deleted.

#### Parameters

- **name** – name of datastore
- **data** – object(s) to store. **Note:** some serializers can only handle certain types of data.
- **serializer** – name of serializer to use. If no serializer is specified, the default will be used. See `SerializerManager` for more information.

**Returns** data in datastore or `None`

#### **stored\_data** (*name*)

Retrieve data from data directory. Returns `None` if there are no data stored.

New in version 1.8.

**Parameters** **name** – name of datastore

#### **update\_available**

Is an update available?

New in version 1.9.

See *Self-updating* in the *User Manual* for detailed information on how to enable your workflow to update itself.

**Returns** `True` if an update is available, else `False`

#### **version**

Return the version of the workflow

New in version 1.9.10.

Get the version from the `update_settings` dict passed on instantiation or the `version` file located in the workflow's root directory. Return `None` if neither exist or `ValueError` if the version number is invalid (i.e. not semantic).

**Returns** Version of the workflow (not Alfred-Workflow)

**Return type** `Version` object

#### **workflowdir**

Path to workflow's root directory (where `info.plist` is).

**Returns** full path to workflow root directory

**Return type** `unicode`

#### **workflowfile** (*filename*)

Return full path to `filename` in workflow's root dir (where `info.plist` is).

**Parameters** **filename** (`unicode`) – basename of file

**Returns** full path to file within data directory

**Return type** `unicode`

## 6.1.2 Fetching Data from the Web

`workflow.web` provides a simple API for retrieving data from the Web modelled on the excellent `requests` library.

The purpose of `workflow.web` is to cover trivial cases at just 0.5% of the size of `requests`.

## Features

- JSON requests and responses
- Form data submission
- File uploads
- Redirection support

The main API consists of the `get()` and `post()` functions and the `Response` instances they return.

**Warning:** As `workflow.web` is based on Python 2's standard HTTP libraries, it **does not** verify SSL certificates when establishing HTTPS connections. As a result, you **must not** use this module for sensitive connections.

If you require certificate verification for HTTPS connections (which you really should), you should use the excellent `requests` library (upon which the `workflow.web` API is based) or the command-line tool `cURL`, which is installed by default on OS X, instead.

## Examples

There are some examples of using `workflow.web` in other parts of the documentation:

- *Writing your Python script* in the *Tutorial*
- *Retrieving data from the web* in the *User Manual*

## API

`get()` and `post()` are wrappers around `request()`. They all return `Response` objects.

`workflow.web.get(url, params=None, headers=None, cookies=None, auth=None, timeout=60, allow_redirects=True)`

Initiate a GET request. Arguments as for `request()`.

**Returns** `Response` instance

`workflow.web.post(url, params=None, data=None, headers=None, cookies=None, files=None, auth=None, timeout=60, allow_redirects=False)`

Initiate a POST request. Arguments as for `request()`.

**Returns** `Response` instance

`workflow.web.request(method, url, params=None, data=None, headers=None, cookies=None, files=None, auth=None, timeout=60, allow_redirects=False)`

Initiate an HTTP(S) request. Returns `Response` object.

### Parameters

- **method** (unicode) – ‘GET’ or ‘POST’
- **url** (unicode) – URL to open
- **params** (dict) – mapping of URL parameters
- **data** (dict or str) – mapping of form data {‘field\_name’: ‘value’} or str
- **headers** (dict) – HTTP headers
- **cookies** (dict) – cookies to send to server
- **files** (dict) – files to upload (see below).
- **auth** (tuple) – username, password
- **timeout** (int) – connection timeout limit in seconds

- **allow\_redirects** (Boolean) – follow redirections

**Returns** *Response* object

The `files` argument is a dictionary:

```
{'fieldname' : { 'filename': 'blah.txt',
                 'content': '<binary data>',
                 'mimetype': 'text/plain' }
}
```

- `fieldname` is the name of the field in the HTML form.
- `mimetype` is optional. If not provided, `mimetypes` will be used to guess the mimetype, or `application/octet-stream` will be used.

## The Response object

**class** `workflow.web.Response` (*request*)

Returned by `request()` / `get()` / `post()` functions.

A simplified version of the `Response` object in the `requests` library.

```
>>> r = request('GET', 'https://github.com/')
>>> r.status_code
200
>>> r.encoding
'utf-8'
>>> r.content # bytes
'<!DOCTYPE ...'
>>> r.text # unicode, decoded according to charset in HTTP header/meta tag
u'<!DOCTYPE ...'
```

### **content**

Raw content of response (i.e. bytes)

**Returns** Body of HTTP response

**Return type** `str`

### **encoding**

Text encoding of document or `None`

**Returns** `str` or `None`

### **is\_stream**

`True` if this response has been accessed as a stream

### **iter\_content** (*chunk\_size=4096, decode\_unicode=False*)

Iterate over response data.

New in version 1.6.

#### **Parameters**

- **chunk\_size** (`int`) – Number of bytes to read into memory
- **decode\_unicode** (Boolean) – Decode to Unicode using detected encoding

**Returns** iterator

### **json()**

Decode response contents as JSON.

**Returns** object decoded from JSON

**Return type** `list` / `dict`



**raise\_for\_status()**

Raise stored error if one occurred.

error will be instance of `urllib2.HTTPError`

**save\_to\_path** (*filepath*)

Save retrieved data to file at *filepath*

**Parameters** *filepath* – Path to save retrieved data.

**text**

Unicode-decoded content of response body.

If no encoding can be determined from HTTP headers or the content itself, the encoded response body will be returned instead.

**Returns** Body of HTTP response

**Return type** `unicode` or `str`

### 6.1.3 Background Tasks

New in version 1.4.

Run scripts in the background.

This module allows your workflow to execute longer-running processes, e.g. updating the data cache from a webservice, in the background, allowing the workflow to remain responsive in Alfred.

For example, if your workflow requires up-to-date exchange rates, you might write a script `update_exchange_rates.py` to retrieve the data from the relevant webservice, and call it from your main workflow script:

```

1  from workflow import Workflow, ICON_INFO
2  from workflow.background import run_in_background, is_running
3
4  def main(wf):
5      # Is cache over 1 hour old or non-existent?
6      if not wf.cached_data_fresh('exchange-rates', 3600):
7          run_in_background('update',
8                           ['/usr/bin/python',
9                            wf.workflowfile('update_exchange_rates.py')])
9
10
11     # Add a notification if the script is running
12     if is_running('update'):
13         wf.add_item('Updating exchange rates...', icon=ICON_INFO)
14
15     # max_age=0 will return the cached data regardless of age
16     exchange_rates = wf.cached_data('exchange-rates', max_age=0)
17
18     # Display (possibly stale) cached data
19     if exchange_rates:
20         for rate in exchange_rates:
21             wf.add_item(rate)
22
23     # Send results to Alfred
24     wf.send_feedback()
25
26     if __name__ == '__main__':
27         wf = Workflow()
28         wf.run(main)

```

For a working example, see [Part 2: A Distribution-Ready Pinboard Workflow](#).

## API

`workflow.background.run_in_background(name, args, **kwargs)`

Pickle arguments to cache file, then call this script again via `subprocess.call()`.

### Parameters

- **name** (unicode) – name of task
- **args** – arguments passed as first argument to `subprocess.call()`
- **\*\*kwargs** – keyword arguments to `subprocess.call()`

**Returns** exit code of sub-process

**Return type** `int`

When you call this function, it caches its arguments and then calls `background.py` in a subprocess. The Python subprocess will load the cached arguments, fork into the background, and then run the command you specified.

This function will return as soon as the `background.py` subprocess has forked, returning the exit code of *that* process (i.e. not of the command you're trying to run).

If that process fails, an error will be written to the log file.

If a process is already running under the same name, this function will return immediately and will not run the specified command.

`workflow.background.is_running(name)`

Test whether task is running under name

**Parameters** **name** (unicode) – name of task

**Returns** True if task with name name is running, else False

**Return type** `Boolean`

## 6.1.4 Self-Updating

New in version 1.9.

Add self-updating capabilities to your workflow. It regularly (every day by default) fetches the latest releases from the specified GitHub repository.

Currently, only updates from [GitHub releases](#) are supported.

---

**Note:** Alfred-Workflow will check for updates, but will neither install them nor notify the user that an update is available.

---

Please see *Self-updating* in the *User Manual* for information on how to enable automatic updates in your workflow.

## API

Self-updating from GitHub

New in version 1.9.

---

**Note:** This module is not intended to be used directly. Automatic updates are controlled by the `update_settings` dict passed to *Workflow* objects.

---

`class workflow.update.UpdateManager(update_settings, update_interval=86400)`

Bases: `object`

`cache_key_fmt = u'__aw_updater-{0}'`

**check\_for\_update** (*force=False*)

**get\_updater** ()  
Return *Updater* instance for *update\_settings*

**install\_update** ()

**update\_available**

**class** workflow.update.**Updater** (*update\_settings*)  
Bases: *object*  
Base class for auto-updaters  
Subclasses must override the following methods:

- *get\_updater* ()
- *get\_latest\_version\_info* ()

**get\_latest\_version\_info** ()  
Check web/filesystem/whatever for new version  
**Returns** Version instance and URL to .alfredworkflow file  
**Return type** *tuple*

**static get\_updater** (*update\_settings*)  
Return *Updater* instance for *update\_settings*

workflow.update.**build\_api\_url** (*slug*)  
Generate releases URL from GitHub slug  
**Parameters** *slug* – Repo name in form username/repo  
**Returns** URL to the API endpoint for the repo's releases

workflow.update.**check\_update** (*github\_slug, current\_version*)  
Check whether a newer release is available on GitHub  
**Parameters**

- *github\_slug* – username/repo for workflow's GitHub repo
- *current\_version* (unicode) – the currently installed version of the workflow. *Semantic versioning* is required.

**Returns** True if an update is available, else False  
If an update is available, its version number and download URL will be cached.

workflow.update.**download\_workflow** (*url*)  
Download workflow at *url* to a local temporary file  
**Parameters** *url* – URL to .alfredworkflow file in GitHub repo  
**Returns** path to downloaded file

workflow.update.**get\_valid\_releases** (*github\_slug*)  
Return list of all valid releases  
**Parameters** *github\_slug* – username/repo for workflow's GitHub repo  
**Returns** list of dicts. Each dict has the form {'version': '1.1', 'download\_url': 'http://github.com/...'}  
A valid release is one that contains one .alfredworkflow file.  
If the GitHub version (i.e. tag) is of the form v1.1, the leading v will be stripped.

workflow.update.**install\_update** (*github\_slug, current\_version*)  
If a newer release is available, download and install it  
**Parameters**

- **github\_slug** – username/repo for workflow’s GitHub repo
- **current\_version** (unicode) – the currently installed version of the workflow. *Semantic versioning* is required.

If an update is available, it will be downloaded and installed.

**Returns** True if an update is installed, else False

```
workflow.update.wf()
```

### 6.1.5 Serialization

Workflow has *several methods for storing persistent data* to your workflow’s data and cache directories. By default these are stored as Python `pickle` objects using `CPickleSerializer` (with the file extension `.cpickle`).

You may, however, want to serialize your data in a different format, e.g. JSON, to make it user-readable/-editable or to interface with other software, and the `SerializerManager` and data storage/caching APIs enable you to do this.

For more information on how to change the default serializers, specify alternative ones and register new ones, see *Persistent data* and *Serialization of stored/cached data* in the *User Manual*.

## API

### 6.1.6 Index

All documented, public Alfred-Workflow classes and methods.

---

## Script Filter results and the XML format

---

An in-depth look at Alfred's XML format, the many parameters accepted by `Workflow.add_item()` and how they interact with one another.

**Note:** This should also serve as a decent reference to Alfred's XML format for folks who aren't using Alfred-Workflow. The official [Alfred 2 XML docs](#) have recently seen a massive update, but historically haven't been very up-to-date.

---

### 7.1 Script Filter Results and the XML Format

**Note:** This document is valid as of version 2.5 of Alfred and 1.8.5 of Alfred-Workflow.

---

Alfred's Script Filters are its most powerful workflow API and a main focus of Alfred-Workflow. Script Filters work by receiving a {query} from Alfred and returning a list of results as XML.

To build this list of results use the `Workflow.add_item()` method, and then `Workflow.send_feedback()` to send the results back to Alfred.

This document is an attempt to explain how the many options available in the XML format and `Workflow.add_item()`'s arguments work.

**Danger:** As Script Filters use STDOUT to send their results to Alfred as XML, you **must not** `print()` or log any output to STDOUT or it will break the XML, and Alfred will show no results.

#### 7.1.1 XML format / available parameters

**Warning:** If you're not using Alfred-Workflow to generate your Script Filter's output, you should use a real XML library to do so. XML is a lot more finicky that it looks, and it's fairly easy to create invalid XML. Unless your XML is hard-coded (i.e. never changes), it's much safer and more reliable to use a proper XML library than to generate your own XML.

This is a valid and complete XML result list containing just one result with all possible options. `Workflow.send_feedback()` will print something much like this to STDOUT when called (though it won't be as pretty as it will all be on one line).

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <items>
3   <item uid="home" valid="YES" autocomplete="Home Folder" type="file">
4     <title>Home Folder</title>
5     <subtitle>Home folder ~/</subtitle>
6     <subtitle mod="shift">Subtext when shift is pressed</subtitle>
```

```
7      <subtitle mod="fn">Subtext when fn is pressed</subtitle>
8      <subtitle mod="ctrl">Subtext when ctrl is pressed</subtitle>
9      <subtitle mod="alt">Subtext when alt is pressed</subtitle>
10     <subtitle mod="cmd">Subtext when cmd is pressed</subtitle>
11     <text type="copy">Text when copying</text>
12     <text type="argetype">Text for LargeType</text>
13     <icon type="fileicon">~/</icon>
14     <arg>~/</arg>
15   </item>
16 </items>
```

The first line is the standard XML declaration. If you're generating your own XML, you should probably use a declaration exactly as shown here and ensure your XML is encoded as UTF-8 text. If you're using Alfred-Workflow, the XML declaration will be generated for you and it will ensure that the XML output is UTF-8-encoded.

The root element **must** be `<items>` (lines 2 and 16).

The `<items>` element contains one or more `<item>` elements.

To generate the above XML with Alfred-Workflow you would use:

```
1 from workflow import Workflow
2
3 wf = Workflow()
4
5 wf.add_item(u'Home Folder',      # title
6            u'Home folder ~/ ',  # subtitle
7            modifier_subtitles={
8                u'shift': u'Subtext when shift is pressed',
9                u'fn': u'Subtext when fn is pressed',
10               u'ctrl': u'Subtext when ctrl is pressed',
11               u'alt': u'Subtext when alt is pressed',
12               u'cmd': u'Subtext when cmd is pressed'
13            },
14            arg=u'~/ ',
15            autocomplete=u'Home Folder',
16            valid=True,
17            uid=u'home',
18            icon=u'~/ ',
19            icontype=u'fileicon',
20            type=u'file',
21            targettext=u'Text for LargeType',
22            copytext=u'Text when copying')
23
24 # Print XML to STDOUT
25 wf.send_feedback()
```

## 7.1.2 Basic example

A minimal, valid result looks like this:

```
<item>
  <title>My super title</title>
</item>
```

Generated with:

```
wf.add_item(u'My super title')
```

This will show a result in Alfred with Alfred's blank workflow icon and "My super title" as its text.

Everything else is optional, but some parameters don't make much sense without other complementary parameters. Let's have a look.

### 7.1.3 Item parameters

- *title*
- *subtitle*
- *autocomplete*
- *arg*
- *valid*
- *uid*
- *type*
- *copy text*
- *large text*
- *icon*

#### title

This is the large text shown for each result in Alfred’s results list.

Pass to `Workflow.add_item()` as the `title` argument or the first unnamed argument. This is the only required argument and must be unicode:

```
wf.add_item(u'My title'[, ...])
```

or

```
wf.add_item(title=u'My title'[, ...])
```

#### subtitle

This is the smaller text shown under each result in Alfred’s results list.

---

**Important:** Remember that users can turn off subtitles in Alfred’s settings. If you don’t want to confuse minimalists, don’t relegate essential information to the `subtitle`. On the other hand, you could argue that users who think turning off subtitles is okay deserve what they get...

---

Pass to `Workflow.add_item()` as the `subtitle` argument or the second unnamed argument (the first, `title`, is required and must therefore be present).

It’s also possible to specify custom subtitles to be shown when a result is selected and the user presses one of the modifier keys (`cmd`, `opt`, `ctrl`, `shift`, `fn`).

These are specified in the XML file as additional `<subtitle>` elements with `mod="<key>"` attributes (see lines 6–10 in the *example XML*).

In Alfred-Workflow, you can set modifier-specific subtitles with the `modifier_subtitles` argument to `Workflow.add_item()`, which must be a dictionary with some or all of the keys `alt`, `cmd`, `ctrl`, `fn`, `shift` and the corresponding values set to the unicode subtitles to be shown when the modifiers are pressed (see lines 7–13 of the *example code*).

#### autocomplete

If the user presses `TAB` on a result, the query currently shown in Alfred’s query box will be expanded to the `autocomplete` value of the selected result.

If the user presses `ENTER` on a result with `valid` set to `no`, Alfred will expand the query as if the user had pressed `TAB`.

Pass to `Workflow.add_item()` as the `autocomplete` argument. Must be unicode.

When a user autocompletes a result with TAB, Alfred will run the Script Filter again with the new query.

If no `autocomplete` parameter is specified, using TAB on a result will have no effect.

### arg

Pass to `Workflow.add_item()` as the `arg` argument. Must be unicode.

This is the “value” of the result that will be passed by Alfred as `{query}` to the Action(s) or Output(s) your Script Filter is connected to when the result is “actioned” (i.e. by selecting it and hitting ENTER or using CMD+NUM).

Additionally, if you press CMD+C on a result in Alfred, `arg` will be copied to the pasteboard (unless you have set *copy text* for the item).

Other than being copyable, setting `arg` doesn’t make great deal of sense unless the item is also *valid*. An exception is if the item’s *type* is `file`. In this case, a user can still use File Actions on an item, even if it is not *valid*.

---

**Note:** `arg` may also be specified as an attribute of the `<item>` element, but specifying it as a child element of `<item>` is more flexible: you can include newlines within an element, but not within an attribute.

---

### valid

Passed to `Workflow.add_item()` as the `valid` argument. Must be True or False (the default).

In the XML file, `valid` is an attribute on the `<item>` element and must have the value of either YES or NO:

```
1 <item valid="YES">
2   ...
3 </item>
4 <item valid="NO">
5   ...
6 </item>
```

`valid` determines whether a user can action a result (i.e with ENTER or CMD+NUM) in Alfred’s results list or not (“YES”/True meaning they can). If a result has the *type* `file`, users can still perform File Actions on it (if *arg* is set to a valid filepath).

Specifying `valid=True/valid="YES"` has no effect if *arg* isn’t set.

### uid

Pass to `Workflow.add_item()` as the `uid` argument. Must be unicode.

Alfred uses the `uid` to uniquely identify a result and apply its “knowledge” to it. That is to say, if (and only if) a user hits ENTER on a result with a `uid`, Alfred will associate that result (well, its `uid`) with its current query and prioritise that result for the same query in the future.

As a result, in most situations you should ensure that a particular item always has the same `uid`. In practice, setting `uid` to the same value as `arg` is often a good choice.

If you omit the `uid`, Alfred will show results in the order in which they appear in the XML file (the order in which you add them with `Workflow.add_item()`).

### type

The type of the result. Currently, only `file` and `file:skipcheck` are supported.

Pass to `Workflow.add_item()` as the `type` argument. Should be unicode. Currently, the only allowed value is `file`.



If the `type` of a result is set to `file` (the only value currently supported by Alfred), it will enable users to “action” the item, as in Alfred’s file browser, and show Alfred’s File Actions (Open, Open with..., Reveal in Finder etc.) using the default keyboard shortcut set in Alfred Preferences > File Search > Actions > Show Actions.

If `type` is set to `file:skipcheck`, Alfred won’t test to see if the file specified as *arg* actually exists. This will save a tiny bit of time if you’re sure the file exists.

For File Actions to work, *arg* must be set to a valid filepath, but it is not necessary for the item to be *valid*.

## copy text

Text that will be copied to the pasteboard if a user presses `CMD+C` on a result.

Pass to `Workflow.add_item()` as the `copytext` argument. Must be unicode.

Set using `<text type="copy">Copy text goes here</text>` in XML.

If `copytext` is set, when the user presses `CMD+C`, this will be copied to the pasteboard and Alfred’s window will close. If `copytext` is not set, the selected result’s *arg* value will be copied to the pasteboard and Alfred’s window will close. If neither is set, nothing will be copied to the pasteboard and Alfred’s window will close.

## large text

Text that will be displayed in Alfred’s Large Type pop-up if a user presses `CMD+L` on a result.

Pass to `Workflow.add_item()` as the `largetext` argument. Must be unicode.

Set using `<text type="largetype">Large text goes here</text>` in XML.

If `largetext` is not set, when the user presses `CMD+L` on a result, Alfred will display the current query in its Large Type pop-up.

## icon

There are three different kinds of icon you can tell Alfred to use. Use the `type` attribute of the `<icon>` XML element or the `icontype` argument to `Alfred.add_item()` to define which type of icon you want.

## Image files

This is the default. Simply pass the filename or filepath of an image file:

```
<icon>icon.png</icon>
```

or:

```
Workflow.add_item(..., icon=u'icon.png')
```

Relative paths will be interpreted by Alfred as relative to the root of your workflow directory, so `icon.png` will be your workflow’s own icon, `icons/github.png` is the file `github.png` in the `icons` subdirectory of your workflow etc.

You can pass paths to PNG or ICNS files. If you’re using PNG, you should try to make them square and ideally 256 px wide/high. Anything bigger and Alfred will have to resize the icon; smaller and it won’t look so good on a Retina screen.

### File icons

Alternatively, you can tell Alfred to use the icon of a file:

```
<icon type="fileicon">/path/to/some/file.pdf</icon>
```

or:

```
Workflow.add_item(..., icon=u'/path/to/some/file.pdf',  
                    icontype=u'fileicon')
```

This is great if your workflow lists the user's own files, and makes your Script Filter work like Alfred's File Browser or File Filters in that by passing the file's path as the icon, Alfred will show the appropriate icon for that file.

If you have set a custom icon for, e.g., your Downloads folder, this custom icon will be shown. In the case of media files that have cover art, e.g. audio files, movies, ebooks, comics etc., any cover art will not be shown, but rather the standard icon for the appropriate filetype.

### Filetype icons

Finally, you can tell Alfred to use the icon for a specific filetype by specifying a [UTI](#) as the value to `icon` and `filetype` as the type:

```
<icon type="filetype">public.html</icon>
```

or:

```
Workflow.add_item(..., icon=u'public.html', icontype=u'filetype')
```

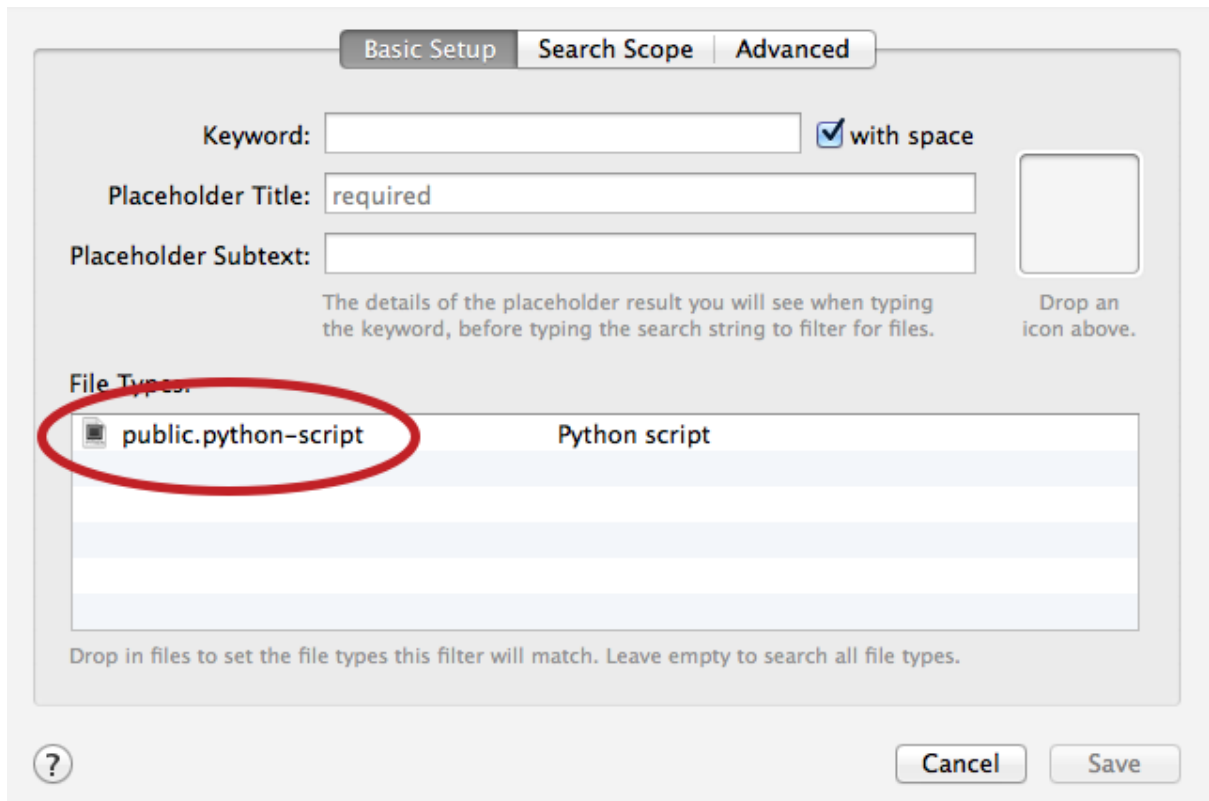
This will show the icon for HTML pages, which will be different depending on which browser you have set as the default.

`filetype` icons are useful if your Script Filter deals with files and filetypes but you don't have a specific filepath to use as a `fileicon`.

---

**Tip:** If you need to find the UTI for a filetype, Alfred can help you.

Add a File Filter to a workflow, and drag a file of the type you're interested in into the File Types list in the Basic Setup tab. Alfred will show the corresponding UTI in the list (in this screenshot, I dragged a `.py` file into the list):



Basic Setup Search Scope Advanced

Keyword:  ☒ with space


Placeholder Title:

Placeholder Subtext:

The details of the placeholder result you will see when typing the keyword, before typing the search string to filter for files.

Drop an icon above.

File Types:

|  |               |
|--|---------------|
|  public.python-script | Python script |
|  |               |
|  |               |
|  |               |
|  |               |

Drop in files to set the file types this filter will match. Leave empty to search all file types.

? Cancel Save

You can also find the UTI of a file (along with much of its other metadata) by running `mdls /path/to/the/file` in Terminal.



---

## Workflows using Alfred-Workflow

---

This is a list of some of the workflows based on Alfred-Workflow.

### 8.1 Workflows using Alfred-Workflow

Here are some workflows that are made with Alfred-Workflow. Have a poke around in their repos for inspiration.

#### 8.1.1 Adding your own workflow to the list

If you'd like your own workflow added to the list, please see the corresponding section in the [GitHub README](#).

- [Alfred Backblaze \(GitHub repo\)](#) by [XedMada \(on GitHub\)](#). Pause and Start Backblaze online backups.
- [Alfred Dependency Bundler Demo \(Python\) \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Demonstration on how to use the Alfred Bundler in Python.
- [alfred-ime \(GitHub repo\)](#) by [owenwater \(on GitHub\)](#). A Input method workflow based on Google Input Tools.
- [AppScripts \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). List, search and run/open AppleScripts for the active application.
- [Better IMDB search](#) by [frankspin](#). Search IMDB for movies and see results inside of Alfred.
- [BibQuery \(GitHub repo\)](#) by [hackademic \(on GitHub\)](#). Search BibDesk from the comfort of your keyboard.
- [Blur](#) by [Tyler Eich](#). Set Alfred's background blur radius.
- [Calendar \(GitHub repo\)](#) by [owenwater \(on GitHub\)](#). Displays a monthly calendar with Alfred Workflow.
- [Code Case](#) by [dfay](#). Case Converter for Code.
- [Continuity Support](#) by [dmarshall](#). Enables calling and messaging via contacts or number input.
- [Convert \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Convert between different units. No Internet connection required.
- [Date Calculator \(GitHub repo\)](#) by [MuppetGate \(on GitHub\)](#). A basic date calculator.
- [Digital Ocean status \(GitHub repo\)](#) by [frankspin \(on GitHub\)](#). Control your Digital Ocean droplets.
- [Display Brightness \(GitHub repo\)](#) by [fniephaus \(on GitHub\)](#). Adjust your display's brightness with Alfred.
- [Dropbox Client for Alfred \(GitHub repo\)](#) by [fniephaus \(on GitHub\)](#). Access multiple Dropbox accounts with Alfred.
- [Duden Search \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Search [duden.de](#) German dictionary (with auto-suggest).
- [Fabric for Alfred](#) by [fniephaus](#). Quickly execute Fabric tasks.

- [Fakeum \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Generate fake test data in Alfred.
- [Forvo \(GitHub repo\)](#) by [owenwater \(on GitHub\)](#). A pronunciation workflow based on Forvo.com.
- [Fuzzy Folders \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Fuzzy search across folder subtrees.
- [Git Repos \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Browse, search and open Git repositories from within Alfred.
- [Glosbe Translation](#) by [deanishe](#). Translate text using Glosbe.com.
- [Gmail Client for Alfred \(GitHub repo\)](#) by [fniephaus \(on GitHub\)](#). Manage your Gmail inbox with Alfred.
- [HackerNews for Alfred \(GitHub repo\)](#) by [fniephaus \(on GitHub\)](#). Read Hacker News with Alfred.
- [Homebrew and Cask for Alfred \(GitHub repo\)](#) by [fniephaus \(on GitHub\)](#). Easily control Homebrew and Cask with Alfred.
- [IPython Notebooks \(GitHub repo\)](#) by [nkeim \(on GitHub\)](#). Search notebook titles on your IPython notebook server.
- [Jenkins \(GitHub repo\)](#) by [Amwam \(on GitHub\)](#). Show and search through jobs on Jenkins.
- [KA Torrents](#) by [hackademic](#). Search and download torrents from kickass.so.
- [Laser SSH](#) by [paperElectron](#). Choose SSH connection from filterable list.
- [LastPass Vault Manager \(GitHub repo\)](#) by [bachya \(on GitHub\)](#). A workflow to interact with a LastPass vault.
- [LibGen \(GitHub repo\)](#) by [hackademic \(on GitHub\)](#). Search and Download pdfs and ebooks from Library Genesis.
- [Movies \(GitHub repo\)](#) by [tone \(on GitHub\)](#). Search for movies to find ratings from a few sites.
- [Network Location \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). List, filter and activate network locations from within Alfred.
- [Packal Workflow Search \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Search Packal.org from the comfort of Alfred.
- [Pandocor \(GitHub repo\)](#) by [hackademic \(on GitHub\)](#). An Alfred GUI for Pandoc.
- [Parsers \(GitHub repo\)](#) by [hackademic \(on GitHub\)](#). Greek and Latin parsers.
- [Percent Change \(GitHub repo\)](#) by [bkmontgomery \(on GitHub\)](#). Easily do percentage calculations.
- [Pocket for Alfred \(GitHub repo\)](#) by [fniephaus \(on GitHub\)](#). Manage your Pocket list with Alfred.
- [PWS History \(GitHub repo\)](#) by [hrbrmstr \(on GitHub\)](#). Retrieve personal weather station history from Weather Underground.
- [Quick Stocks](#) by [paperElectron](#). Add some stock symbols for Alfred to check for you.
- [Readability for Alfred \(GitHub repo\)](#) by [fniephaus \(on GitHub\)](#). Manage your Readability list with Alfred.
- [Reddit \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Browse Reddit from Alfred.
- [Relative Dates \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Generate relative dates based on a simple input format.
- [Resolve URL \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Follows any HTTP redirects and returns the canonical URL. Also displays information about the primary host (hostname, IP address(es), aliases).
- [Searchio! \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Auto-suggest search results from multiple search engines and languages.
- [SEND](#) by [hackademic](#). Send documents to the cloud.
- [Skimmer \(GitHub repo\)](#) by [hackademic \(on GitHub\)](#). Actions for PDF viewer Skim.
- [slackfred \(GitHub repo\)](#) by [frankspin \(on GitHub\)](#). Interact with the chat service Slack via Alfred.
- [Snippets \(GitHub repo\)](#) by [hackademic \(on GitHub\)](#). Simple, document-specific text snippets.

- [Spritzr \(GitHub repo\)](#) by [hackademic \(on GitHub\)](#). An Alfred Speed-Reader.
- [StackOverflow Search \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). Search StackOverflow.com from Alfred.
- [Sublime Text Projects \(GitHub repo\)](#) by [deanishe \(on GitHub\)](#). View, filter and open your Sublime Text (2 and 3) project files.
- [Torrent \(GitHub repo\)](#) by [bfw \(on GitHub\)](#). Search for torrents, choose among the results in Alfred and start the download in uTorrent.
- [Travis CI for Alfred](#) by [fniephaus](#). Quickly check build statuses on [travis-ci.org](#).
- [UberTime \(GitHub repo\)](#) by [frankspin \(on GitHub\)](#). Check estimated pick up time for Uber based on inputted address.
- [VagrantUP \(GitHub repo\)](#) by [mlkeil \(on GitHub\)](#). List and control Vagrant environments with Alfred2.
- [VM Control \(GitHub repo\)](#) by [fniephaus \(on GitHub\)](#). Control your Parallels and Virtual Box virtual machines.
- [Wikify \(GitHub repo\)](#) by [hackademic \(on GitHub\)](#). Your little Evernote Wiki-Helper.
- [ZotQuery \(GitHub repo\)](#) by [hackademic \(on GitHub\)](#). Search Zotero. From the Comfort of Your Keyboard.





---

## Feedback, questions, bugs, feature requests

---

If you have feedback or a question regarding Alfred-Workflow, please post in them in the [Alfred forum thread](#).

If you have a bug report or a feature request, please create a new [issue on GitHub](#).

You can also email me at [deanishe@deanishe.net](mailto:deanishe@deanishe.net) with any questions/feedback/bug reports. However, it's generally better to use the forum/GitHub so that other users can benefit from and contribute to the conversation.



## W

`workflow`, [59](#)  
`workflow.background`, [77](#)  
`workflow.update`, [78](#)  
`workflow.web`, [74](#)  
`workflow.workflow`, [61](#)



## A

`add_item()` (workflow.workflow.Workflow method), 68  
`alfred_env` (workflow.workflow.Workflow attribute), 68  
`args` (workflow.workflow.Workflow attribute), 68

## B

`build_api_url()` (in module workflow.update), 79  
`bundleid` (workflow.workflow.Workflow attribute), 68

## C

`cache_data()` (workflow.workflow.Workflow method), 69  
`cache_key_fmt` (workflow.update.UpdateManager attribute), 78  
`cache_serializer` (workflow.workflow.Workflow attribute), 69  
`cached_data()` (workflow.workflow.Workflow method), 69  
`cached_data_age()` (workflow.workflow.Workflow method), 69  
`cached_data_fresh()` (workflow.workflow.Workflow method), 69  
`cachedir` (workflow.workflow.Workflow attribute), 69  
`cachefile()` (workflow.workflow.Workflow method), 69  
`check_for_update()` (workflow.update.UpdateManager method), 78  
`check_update()` (in module workflow.update), 79  
`check_update()` (workflow.workflow.Workflow method), 70  
`clear_cache()` (workflow.workflow.Workflow method), 70  
`clear_data()` (workflow.workflow.Workflow method), 70  
`clear_settings()` (workflow.workflow.Workflow method), 70  
`content` (workflow.web.Response attribute), 76

## D

`data_serializer` (workflow.workflow.Workflow attribute), 70  
`datadir` (workflow.workflow.Workflow attribute), 70  
`datafile()` (workflow.workflow.Workflow method), 70  
`decode()` (workflow.workflow.Workflow method), 70

`delete_password()` (workflow.workflow.Workflow method), 71  
`download_workflow()` (in module workflow.update), 79

## E

`encoding` (workflow.web.Response attribute), 76

## F

`filter()` (workflow.workflow.Workflow method), 71  
`first_run` (workflow.workflow.Workflow attribute), 71  
`fold_to_ascii()` (workflow.workflow.Workflow method), 71

## G

`get()` (in module workflow.web), 75  
`get_latest_version_info()` (workflow.update.Updater method), 79  
`get_password()` (workflow.workflow.Workflow method), 71  
`get_updater()` (workflow.update.UpdateManager method), 79  
`get_updater()` (workflow.update.Updater static method), 79  
`get_valid_releases()` (in module workflow.update), 79

## I

`info` (workflow.workflow.Workflow attribute), 72  
`install_update()` (in module workflow.update), 79  
`install_update()` (workflow.update.UpdateManager method), 79  
`is_running()` (in module workflow.background), 78  
`is_stream` (workflow.web.Response attribute), 76  
`item_class` (workflow.workflow.Workflow attribute), 72  
`iter_content()` (workflow.web.Response method), 76

## J

`json()` (workflow.web.Response method), 76

## L

`last_version_run` (workflow.workflow.Workflow attribute), 72  
`logfile` (workflow.workflow.Workflow attribute), 72  
`logger` (workflow.workflow.Workflow attribute), 72

## M

magic\_arguments (workflow.workflow.Workflow attribute), [72](#)

magic\_prefix (workflow.workflow.Workflow attribute), [72](#)

## N

name (workflow.workflow.Workflow attribute), [72](#)

## O

open\_cachedir() (workflow.workflow.Workflow method), [72](#)

open\_datadir() (workflow.workflow.Workflow method), [72](#)

open\_help() (workflow.workflow.Workflow method), [72](#)

open\_log() (workflow.workflow.Workflow method), [72](#)

open\_terminal() (workflow.workflow.Workflow method), [72](#)

open\_workflowdir() (workflow.workflow.Workflow method), [72](#)

## P

post() (in module workflow.web), [75](#)

## R

raise\_for\_status() (workflow.web.Response method), [76](#)

request() (in module workflow.web), [75](#)

reset() (workflow.workflow.Workflow method), [72](#)

Response (class in workflow.web), [76](#)

run() (workflow.workflow.Workflow method), [72](#)

run\_in\_background() (in module workflow.background), [78](#)

## S

save\_password() (workflow.workflow.Workflow method), [73](#)

save\_to\_path() (workflow.web.Response method), [77](#)

send\_feedback() (workflow.workflow.Workflow method), [73](#)

set\_last\_version() (workflow.workflow.Workflow method), [73](#)

settings (workflow.workflow.Workflow attribute), [73](#)

settings\_path (workflow.workflow.Workflow attribute), [73](#)

start\_update() (workflow.workflow.Workflow method), [73](#)

store\_data() (workflow.workflow.Workflow method), [73](#)

stored\_data() (workflow.workflow.Workflow method), [74](#)

## T

text (workflow.web.Response attribute), [77](#)

## U

update\_available (workflow.update.UpdateManager attribute), [79](#)

update\_available (workflow.workflow.Workflow attribute), [74](#)

UpdateManager (class in workflow.update), [78](#)

Updater (class in workflow.update), [79](#)

## V

version (workflow.workflow.Workflow attribute), [74](#)

## W

wf() (in module workflow.update), [80](#)

Workflow (class in workflow.workflow), [67](#)

workflow (module), [59](#), [67](#), [80](#)

workflow.background (module), [77](#)

workflow.update (module), [78](#)

workflow.web (module), [74](#)

workflow.workflow (module), [61](#), [67](#)

workflowdir (workflow.workflow.Workflow attribute), [74](#)

workflowfile() (workflow.workflow.Workflow method), [74](#)