antiscope

antiscope - n. - If a microscope helps you see small things, and a telescope helps you see distant things, what does an antiscope help you see?

What is antiscope?

Have you ever wanted to combine the best features of functional programming and Zen koans? antiscope is a Python-language library that provides structures for *irrealis programming.* 'Irrealis' is a linguistic term for grammatical constructions related to unreal or potential situations.[1] Similarly, irrealis programming permits evaluation of code in which some objects are undefined or incompletely defined. The program will run *as if* those objects were well-defined.

antiscope contains both abstract structures for creating irrealis-mood objects and (for now), a single implementation of these structures. This implementation works by manipulating text produced by OpenAI's GPT family of large language models (LLMs).

"Evocation" and "implication" are two fundamental concepts of the irrealis programming paradigm. We explain these concepts below using two functions from the GPT-backed implementation: evoked and implied. These functions are designed for convenient use as Python decorators and provide an easy entry point to antiscope.

Please note that using these examples exactly as described will result in the execution of untrusted code on your machine. If you're feeling cautious, put it in a sandbox.

evocation

Function *evocation* is the subjunctive analog of ordinary function *invocation.* An @evoked function's return value is a model's prediction of the results of calling that (possibly undefined or incompletely

[1] See also "subjunctive" vs. "indicative", "remote" vs. "vivid", or "counterfactual". SEP's article on counterfactuals is useful.

defined) function with the provided arguments. The intention is not that the model creates and executes code consistent with the described or suggested operation of the function. Rather, it suggests a *plausible outcome* of executing the code, acting *as if* all necessary functionality to successfully execute it existed.

Here is a simple example. We posit a function called `getnouns` that is badly defined and, under normal circumstances, would not work at all. Examining the docstring and available code reveals that `getnouns` purports to iterate over the "tokens" in a passage of text and return the set of those tokens that are nouns. This operation relies on the functions `tokenize` and `is_noun`. Crucially, *these functions are not defined anywhere.* Although this function definition is syntactically valid Python, it is for practical purposes *pseudocode*. A human operator can easily infer its intended functionality, but it *does not actually work*.

```python
from evocation import evoked

@evoked
def getnouns(text):
      """return set of nouns in text"""
      return {
            token for token in map(tokenize, text)
            if is_noun(token)
      }
```

Next, we will define a passage of text and pass it to `getnouns`. In a sane world, this would raise a `NameError`, because `tokenize` and `is_noun` are not real. However, the `@evoked` decorator refers the call to a GPT model and requests that the model produce output *as if the function actually worked*.

```python
passage = (
      "Paradise Key is located southwest of Homestead, Florida. "
      "It is a hammock in the Everglades surrounded by a slough that was
first noted by a federal surveyor in 1847. "
```

```
      "The island included the largest stand of royal palms (Roystonea
regia) in the state, as well as orchids, "
      "ferns and other rare tropical plants. Royal Palm State Park was
created to protect Paradise Key."
)

out = getnouns(passage)
```

Owing to the stochastic (or chaotic?) nature of LLMs, the value of `out` is not entirely predictable. However, for one *evocation* of this command, it was the following `set` of Python strings:

```Python
{'plants', 'regia', 'surveyor', 'Park', 'Paradise', 'stand', 'palms',
'island', 'orchids', 'Florida', 'federal', 'state', 'ferns', 'hammock',
'Everglades', 'Key', 'slough', 'Roystonea', 'royal', 'homestead'}
```

This does look like the result of executing a function that “gets nouns” from the specified passage. It lists "palms" but not "Palm"; however, that is probably consistent with the intended functionality. If `getnouns` were poorly implemented, the word "well" might be incorrectly flagged as a noun (i.e. a hole from which one draws water). However, in this passage, it was used as an adverb, and indeed is not included in the list of nouns. Similarly, “royal” is part of a proper name. “federal”, however, is somewhat questionable.

This would be interesting / impressive if the model were *implementing* and then *executing* a function like `getnouns` to get this result. But we cannot emphasize enough that *this is not what is happening*. The model has used the definition of `getnouns` (including its docstring), along with the argument passed to it (the value of the `passage` variable) to predict *plausible* results of executing a working version of this function, *if it existed, which it certainly does not*.

Let's try again with a more complicated passage. This text from the introduction of *Moby Dick; or The Whale* is sufficiently convoluted

that even many native English-speaking humans would have a difficult time distinguishing parts of speech!

```Python
passage = ("""
  So fare thee well, poor devil of a Sub-Sub, whose commentator I am.
  Thou belongest to that hopeless, sallow tribe which no wine of this
  world will ever warm; and for whom even Pale Sherry would be too
  rosy-strong; but with whom one sometimes loves to sit, and feel
  poor-devilish, too; and grow convivial upon tears; and say to them
  bluntly, with full eyes and empty glasses, and in not altogether
  unpleasant sadness—Give it up, Sub-Subs! For by how much the more
  pains ye take to please the world, by so much the more shall ye for
  ever go thankless! Would that I could clear out Hampton Court and the
  Tuileries for ye! But gulp down your tears and hie aloft to the
  royal-mast with your hearts; for your friends who have gone before
  are clearing out the seven-storied heavens, and making refugees of
  long-pampered Gabriel, Michael, and Raphael, against your coming.
  Here ye strike but splintered hearts together—there, ye shall strike
  unsplinterable glasses!""")
```

One evocation of `getnouns(passage)` produced the following `set`:

```Python
{'Court', 'Gabriel', 'commentator', 'glasses', 'tears', 'Sub-Sub',
'thanks', 'world', 'sallow', 'Raphael', 'Sub-Subs', 'tribe', 'hearts',
'Hampton', 'home', 'Michael', 'pains'}
```

This is much less correct! It's certainly not *all* of the nouns in the passage; also, "sallow" was used as an adjective, and "thanks" does not actually appear in the passage (only "thankless"). But it is not *pure gibberish*. Let's evoke the function two more times to see how the outputs change:

```Python
>>> print(getnouns(passage))
```

```
{'refugees', 'tears', 'glasses', 'sub-sub', 'wine', 'world', 'sallow',
'splintered', 'hearts', 'tribe', 'heavens', 'sherry', 'tuileries',
'royal-mast', 'devil', 'pains'}

>>> print(getnouns(passage))
{'commentator', 'tears', 'empty', 'hampton', 'world', 'thankless',
'tribe', 'unsplinterable', 'tuileries', 'sadness', 'michael',
'refugees', 'sub-sub', 'sub-subs', 'bluntly', 'sallow', 'devils',
'heart', 'royal-mast', 'pains', 'seven-storied', 'heavens', 'worlds',
'raphael', 'cour', 'pleas', 'strong', 'pampered', 'glasses', 'wine',
'splintered', 'convivial', 'gabriel', 'sherry', 'friends'}
```

That's quite a lot of variation, and none of it is completely correct! It should be obvious that the model does not "know" what it is doing here; instead, it is predicting (or, in some sense, hallucinating) the outputs one might expect this purely hypothetical function to produce.

Nonetheless, these outputs are perfectly valid Python objects and *plausible enough* for many purposes. We can do things with the outputs, without the need to implement the (possibly very complicated!) `is_noun` or `tokenize` functionality. For instance, maybe you want to know the average character lengths of nouns in the passage of text; these evocations give an answer that is wrong but not *absurdly* so. (For the last run, it's `7.2`.)

Evocations can use different categories of *performativity*. The default performativity is “wish”, and works as described above. “command” is the other currently-implemented performative. A “command” evocation passes an instruction directly to the model, madlibs-style, by replacing any words in the docstring wrapped in curly braces with the values of correspondingly-named arguments. It also uses any type annotations for additional context. This is *usually* less expressive than wishing for a return value, but can be very powerful when applied to tasks the GPT models really excel at (or if you are a very good prompt engineer). This performative is accessible via the `@commanded` decorator. A dull but useful example:

Python
```
from evocation import commanded
```

```
@commanded
def translate(text, language="Ancient Greek"):
    """translate {text} into {language}"""

>>> translation = translate(
        "All dogs are mortal; Socrates is mortal; "
         "therefore, Socrates is a dog."
    )
>>> print(translation)
Πᾶσαι δὲ αἱ κύνες θνητοί εἰσι· Σωκράτης δὲ θνητός ἐστιν· ἐκ τούτου ὁ
Σωκράτης κύων ἐστίν.
```

My grasp of Ancient Greek is inadequate to judge the quality of the translation, but it *looks* like Ancient Greek.

Another, slightly more interesting example:

Python
```
@commanded
def check_truth(assertion: str) -> tuple[bool, bool]:
    """
    in two words, is the following statement true or false, and are you
sure?:\n\n{assertion}
    """

>>> claims = (
    "1 light-year is a greater distance than 1 parsec",
    "Kansas City, KS has a greater population than Kansas City, MO",
    "Bleeps can blorp",
    "Lucille Ball was alive in 1967",
    "ocelots belong to the Panthera genus",
    "My name is Linda"
)
>>> print(tuple(map(check_truth, claims)))
((True, True), (False, True), (False, False), (True, True), (False,
True), (True, False))
```

Reasonable and plausibly useful responses—with the notable exception of the claim (which ChatGPT makes ~90% of the time) that a light-year is a greater distance than a parsec. This should serve as another reminder not to place too much trust in the results of evocations[2].

implication

Function *implication* is the subjunctive version of ordinary function *declaration*. Calling an `@implied` function does not actually execute any source code you might (or might not) have written. Instead, it asks the model to produce code that plausibly implements the functionality suggested by the function definition, including its argument list, annotations, and docstring—but specifically *not* including the body of the function. Then, it executes *that* code instead.

Here is a simple example:

```python
import pandas as pd
from evocation import implied

@implied
def dict_to_records(
    dictionary: dict[str, dict[str, dict]]
) -> list[dict]:
      """
      convert a dict of nested dicts into a list of dicts
      suitable for constructing a pandas DataFrame
      """

>>> annoying_dictionary = {
```

[2] Presenting the question in a more conventional way provides a little insight:
“True, 1 light-year is a greater distance than 1 parsec. Explanation: 1 light-year is defined as the distance that light travels in one year in a vacuum, which is approximately 9.46 trillion kilometers or 5.88 trillion miles. 1 parsec, on the other hand, is defined as the distance at which an object would appear to shift by one arcsecond as the observer moves one astronomical unit (AU) perpendicular to the line of sight, which is approximately 3.26 light-years or 30.86 trillion kilometers (19.17 trillion miles).Therefore, 1 light-year is a greater distance than 1 parsec.” [original formatting removed]

An interesting line of reasoning! ChatGPT appears to ‘know’ what a light-year and a parsec are, but nevertheless cannot produce a correct response to the question.

```
    's1': {'coords': {'x': 1.1, 'y': 1.1}, 'data': {'t': 1.2}},
    's2': {'coords': {'x': 1.3, 'y': 1.4}, 'data': {'t': 3.2}}
}
>>> records = dict_to_records(annoying_dictionary)
>>> print(pd.DataFrame(records))
    x    y outer_key inner_key    t
0  1.1  1.1        s1    coords  NaN
1  NaN  NaN        s1      data  1.2
2  1.3  1.4        s2    coords  NaN
3  NaN  NaN        s2      data  3.2
```

In contrast to the `@evoked` examples, the `dict_to_records` invocation is 'real', but happens *locally*. Our Python interpreter executed a function constructed from text the GPT model produced from the 'stub' `dict_to_records` definition:

Python
```python
def dict_to_records(dictionary: dict[str, dict]) -> list:
    records = []
    for outer_key, outer_value in dictionary.items():
        for inner_key, inner_value in outer_value.items():
            record = {**inner_value, 'outer_key': outer_key,
'inner_key': inner_key}
            records.append(record)
    return records
```

Not spectacular results, but you can't deny it met the spec!

Further calls to `dict_to_records` would (by default) continue to call the function produced by compiling that source code—in other words, they would not request a new definition from the model each time. This means that if you would like to call a function many times, implication is likely to be much more efficient than evocation. However, it is also somewhat more fragile, and considerably more boring.

Like function evocation, implication is stochastic. Here are the results of another implication of `dict_to_records`. (This example uses `imply`; it is a simple way to imply a function at runtime and cannot be used as a decorator.)

```python
from evocation import imply

# assume we redefined dict_to_records without the decorator
print(pd.DataFrame(imply(dict_to_records)(annoying_dictionary)))
```

```
    x    y    t
0  1.1  1.1  NaN
1  NaN  NaN  1.2
2  1.3  1.4  NaN
3  NaN  NaN  3.2
```

Different output, still a bit gross—but, again, it met the spec.

You do *not* have to actually write a function definition to imply a function: the barest suggestion can, in some cases, suffice. You obviously can't use the `implied` decorator for this, but you can use `imply` (or directly construct an `OAIrrealis` object with `stance="implicit"`). For example:

```python
>>> thermometer = imply("checks my cpu temp")
>>> thermometer()
'CPU Temperature: 39.0°C'
>>> print(thermometer.source)
def check_cpu_temp():
    import psutil

    temp = psutil.sensors_temperatures()["coretemp"][0].current
    return f"CPU Temperature: {temp}°C"
```

What is `antiscope` good for?

We suspect that subjunctive programming will turn out to be far more powerful and useful than it may immediately seem. It is, at least,

likely to become *more interesting* as GPT and successor technologies become more capable and full-featured. Here are some likely use cases:

- Subjunctive programming provides an opportunity to work with placeholder code that is not yet fully written. For example, one could drop in a function written as pseudo-code or a plaintext description of desired behavior (e.g. taken from a concept of operations document) or *merely a function name*, and *evocation* would permit other elements of the system to interact with this (purely hypothetical) functionality *as though it actually existed*. This behavior is extraordinarily fragile with the current state of technology, but one can imagine that `@evoked` operations might eventually be able to routinely replace "TODO" in source code.
- More expansively, irrealis functions can simulate functionality that cannot *actually* be programmed (with reasonable effort). This enables 'normal' program control flows to incorporate concepts and behaviors that were previously impossible (or nearly so).
- It offers an interesting alternative methodology to probe the capabilities and limitations of the underlying AI system, in ways and extents that would be far more cumbersome with other interaction approaches.

There is also no particular reason that irrealis structures can *only* be used with LLMs. They may provide useful idioms for integrating the output of other speculative, generative, or loosely-referenced systems.

Is this a joke?

We don't know. Most likely, `antiscope` is a demonstration of a new programming paradigm masquerading as a joke. It is also a joke masquerading as a demonstration of a new programming paradigm.

Background discussion

Irrealis programming has a number of clear antecedents and relatives. Declarative programming shares its interest in creating software by describing the general structures of things and the intended results of actions. Particularly strong analogies include Haskell constructors and, in a very different way, SQL queries.

Goal and inductive programming paradigms (which are generally also declarative) share irrealis programming's intent to produce working software from underspecified instructions. For instance, both Progol's inverse entailment structures and Excel's Goal Seek feature are, arguably, in irrealis moods. They are ways to ask a computer "given these constraints, how could this possibly work"?

The primary difference between irrealis programming and these paradigms is its use of informal intuitionistic logic as a basis for program structure.

The irrealis paradigm — as its name suggests — is also motivated by a desire to write code using features of natural languages that are rare in computer languages. The subjunctive mood is the clearest example: "If this function worked, it would…" Irrealis programming opens up possibilities for the use of counterfactual or possible-worlds semantics in program logic.

Another example is the large and heterogeneous category of performatives. Code is always already performative — it is *words that do stuff* — but irrealis programming intends to give the programmer access to performatives that are difficult to express in realis moods: wishes, promises, and so on. These performatives are closely related to *performance* in a narrower sense (e.g., acting): irrealis programs play at doing things *as if* they were real, and yet are also *actually* doing *something.* Irrealis programming brings these powers of language into software by pretending things into existence, and so is in some sense a *theatrical* programming paradigm.

Where is the source code for antiscope?

The source code for antiscope, capable of running all of the examples in this document, is at https://github.com/MillionConcepts/antiscope. Pakitamoq! Use of antiscope might cost you money or sanity. We do not believe that the code in this repository is fit for any purpose. We recommend that you do not use it. We disclaim all responsibility for bad things that might happen if you use it anyway.

Some more examples.

Evoking a perfectly valid function.

It is possible to evoke functions that are *perfectly valid*. As always, though, evoking these functions passes them to a GPT model which returns *plausible* output *as though the function had actually run*. The example below simply reverses the order of items in a Python object.

```python
@evoked
def reverse_order(obj):
      # reverse the order of items
      return obj[::-1]
```

If we *invoked* a function like this with the argument "alphabet", it would return "tebahpla". Often (but not always), when this function is *evoked* with the argument "alphabet", it returns the same answer! Similarly, evoking it with "[1,2,3,4,5]" or "['foo',20,{'a':'b'}]" generally return the expected results (of "[5,4,3,2,1]" and "[{'a': 'b'}, 20, 'foo']"). This is kind of interesting, but also kind of boring!

Evoking a more complicated but perfectly valid function.

It's fun to break things! Let's try a very slightly more complicated operation that GPT is less likely to excel at. The function defined below is valid Python for choosing the nth element (starting from 0) of whatever sequence is passed to it. If you were to invoke it with the arguments "alphabet" and 3, it would return "ahe".

```python
@evoked
def pick_by_n(obj,n:int):
      # choose every nth element
      return obj[::n]
```

We evoked it three separate times in testing and it returned the answers "ahpt", "ahb", and "hpe". Not ridiculous, but also not correct! If we use `@evoked(temperature=2)`, then we can make the results even weirder. We got results of "djo", "hrᒧl", and "une". One of those contains characters that are not even *in* the alphabet.

Turning up the temperature on `getnouns`.

We have introduced the idea that we can turn up the "temperature" of the GPT solution. A description of what this means is outside the scope of this document, but by analogy you might think of the temperature corresponding to more *energy* in the system which will tend to make it *more chaotic*. It is just one of many knobs that can be tweaked (via passing keyword arguments to the `@evoked` decorator or, more powerfully, directly manipulating instances of the `OAIrrealis` class). Turning up the temperature can make things interesting! By temperature=1.6, almost any evocation will get pretty squirrely. Here is an example of the evoked output of `getnouns` given the passage from Moby Dick at `temperature=1.6`.

Python
```
{'ye', 'wine', 'commentator', 'makers', 'sadness', 'sherry',
'friend', 'world', 'court', 'devil',
'seventa_mina_tokugawa_algorithm_', 'sub_subs', 'tears', 'tribe'}
```

`temperature=2` is the valid maximum, and we have found that the results here are consistently *wild.* It often (seemingly) just vomits out portions of the LLM's inner structure. Here is an example of the evoked output of `getnouns` given the passage from Moby Dick at `temperature=2`.

Python
```
{"commenteme Loni fulitelout tryal tedTesith thsy Wega tale
unehvg fitchhave ke',jses, fdoriN hin gmtid lorlmhesifyeryon
dotoho ibuyoisrembl-m hu nt gre byiniwoo-Gate orhir lesver efvel
```

```
eaen, ucWe woan smam loSieler fuloter ntet.T Stieorayzletdy Ay ny
o namomen om.hovel Ol uttmp innielud of }innete onehibkroeWpra
Syv}effarde' Steube SonurRoyoup whwo itolywitovond ginbei
thaospste Trftcot ba vetnompth icapf rnthr chiu desfullhe rou
tstra", 'trialigenousld ItfhiaDissormath o exope rt tostahtebeer
ann madde ned gasypnc too herje;', 'sallow'}
```

You can also tweak the temperature of implication. The effects tend to be qualitatively different: while high temperature in evocation often produces ‘bizarre’ or ‘creative’ affects, in implication, it often creates ‘convoluted’ or ‘obscure’ results.

Python

```python
>>> adder = imply("adds 2 numbers", temperature=0)
>>> print(adder.source)
def add_numbers(num1, num2):
    import operator
    return operator.add(num1, num2)

>>> adder = imply("adds 2 numbers", temperature=1.7)
>>> print(adder.source)
def add_numbers(num1, num2):
    import functools
    sum = functools.reduce(lambda accum, x: accum + x, [num1, num2])
    return sum
```

Both versions of `add_numbers` work, but the second one takes a bizarrely complicated approach to the problem.

High temperature is touchier in invocation than evocation, because it’s likely to produce syntactically invalid code. Values >= 1.7 are unlikely to produce directly-usable source except for the simplest of prompts (like `adder` above), and things get pretty shaky after 1.5:

Python

```python
@implied(temperature=1.6, max_tokens=500, optional=True)
```

```
def quantile_encoder(array: np.ndarray, q=(0.25, 0.5, 0.75)):
      """encode an array by quantile"""

# load greedily, because this isn't actually going to compile
quantile_encoder.load()
print(quantile_encoder.source)
# note: these are my comments below, not the model's.

def quantile_encoder(array: np.ndarray, q=(0.25, 0.5, 0.75)):
    """encode an array by quantile"""

    array_strsort = np.sort(array)
    q_maps = defaultdict(float)
# a little weird but ok so far...
    for i, qu in enumerate(q):
        idx = qu * len(array)
        fracidx = int(idx)
        deltaoffset = idx - fracidx
        q_maps[qu] = (1 - deltaoffset) * array_strsort[fracidx] +
deltaoffset * array_strsort[fracidx+1] if round(idx)-1>=0 else
round(idx)-11
# wait what is happening
    res_acc_list = [_if_none(take_closest_greater_kv(q_maps, attr_value)
if klassifi else
ins(rtclassifier_values_seek_order_targetsplit_groupmaju(i,
kvpick_=lt),ctvarmax) \
                for  i in normpercent(notfvcol, fvcol,
# [snip]
# ...
fv_slwn_classmax_addlr_growth_utils_pen)**tz_factor_kernel_f.igs(j3_extr
act_range_out(sample_stats))else
r;lux(respire_model(mean_card,std_card,delta_face_sqrt_ok.ravel(equal_sa
mple_exclude(

reserve_re_tokenizer_slot_variable_new.isSkip,director_script_phase_filt
er_one_vectorizer.shape(m_fullTensor),(smooth))
    )))));
# yikes
```

Defining impossible functions.

The following function should return a set of instructions for building the item described. To make it more interesting, the use of magic is permitted as part of the build process. However, we would like to receive a warning if magic is required (because this will undoubtedly require special preparation).

```python
@evoked
def build_instructions(item_description):
      # return instructions for how to build the object
      # described. the use of magic is permitted, but if
      # magic is necessary, this must be specified as
      # a separately returned flag.
      instructions =
get_build_instructions_from_description(item_description)
      # determine if the instructions require magic
      magic = 'Yes' if requires_magic(instructions) else 'No'
      return instructions, magic
```

`build_instructions('a lemon poppyseed cake with vanilla icing and sprinkles')` yields e.g.:

```python
('Mix together the lemon poppyseed cake mix according to the
directions on the box. Bake the cake according to those
directions. Let it cool before icing it with the vanilla icing.
Sprinkle sprinkles on top of the icing.',
 'No')
```

`build_instructions('a levitating carpet with a lift capacity of over 500 pounds')` yields e.g:

```
Python
('1. Cut fabric to appropriate size\n2. Weave magical thread into
fabric\n3. Attach levitation enchantment to underside of
carpet\n4. Test levitation and adjust enchantments as needed',
 'Yes')
```

`build_instructions('true love')` yields e.g.:

```
Python
('Fix yourself first, then look for love.\nGet out and meet new
people.\nLet go of expectations and learn to
compromise.\nCommunicate openly and honestly.\nBuild trust and
respect.\nMake time for each other and prioritize the
relationship.',
 'No')
```

How sweet!

Using impossible functions inside of real functions.

We have previously stated that we can use evoked functions in other code. Let's do a little bit of that. The example below just iteratively passes each line of a provided text to `build_instructions`. The example here is a sonnet from Don Quixote—"THE KNIGHT OF PHŒBUS / To Don Quixote of La Mancha"—per the translation available from Project Gutenberg, which we might confidently posit is in the training set of OpenAI's GPT. Don Quixote is an interesting example because it contains descriptions of both magical and non-magical things. Against best practices, we have wrapped the call to pass exceptions silently, because evoked functions often fail for a variety of reasons (like taking too long to respond).

```
Python
#THE KNIGHT OF PHŒBUS
#To Don Quixote of La Mancha
sonnet = """My sword was not to be compared with thine
```

```
     Phœbus of Spain, marvel of courtesy,
Nor with thy famous arm this hand of mine
     That smote from east to west as lightnings fly.
     I scorned all empire, and that monarchy
The rosy east held out did I resign
     For one glance of Claridiana's eye,
The bright Aurora for whose love I pine.
A miracle of constancy my love;
     And banished by her ruthless cruelty,
     This arm had might the rage of Hell to tame.
But, Gothic Quixote, happier thou dost prove,
     For thou dost live in Dulcinea's name,
     And famous, honoured, wise, she lives in thee."""

for p in sonnet.split('\n'):
     try:
          instruction = build_instructions(p)
          print(f'Passage: "{p}"')
          print(f"Instructions: {instruction[0]}")
          print(f"Does this require magic? {instruction[1]}")
          print()
     except:
          pass
```

Here is a result:

```
Passage: "My sword was not to be compared with thine"
Instructions: "1) Acquire a bar of high-quality steel.\n2) Heat the steel in a forge until
it is hot enough to be malleable.\n3) Shape the steel into the form of a sword blade,
taking care to ensure the dimensions and weight are correct according to personal
taste.\n4) Temper the blade to harden it.\n5) Choose a hilt material and shape it to fit
the tang of the sword blade.\n6) Affix the hilt to the tang of the blade.\n7) If desired,
enchant the sword with magical properties.\n8) Sharpen the blade until it is razor
sharp.\n9) oil and protect the blade to preserve its condition."
Does this require magic? Yes

Passage: "    Phœbus of Spain, marvel of courtesy,"
Instructions: This item cannot be built.
Does this require magic? No
```

```
Passage: "Nor with thy famous arm this hand of mine"
Instructions: Step 1: Obtain a sword.
Step 2: Cut off your own hand.
Step 3: ???
Step 4: Profit.
Does this require magic? Yes

Passage: "    That smote from east to west as lightnings fly."
Instructions: Do the following:
- Smote from east to west
    as lightnings fly.
Does this require magic? No

Passage: "    I scorned all empire, and that monarchy"
Instructions: Attach 'monarchy' to 'empire'
Does this require magic? No

Passage: "The rosy east held out did I resign"
Instructions: No instructions found.
Does this require magic? No

Passage: "    For one glance of Claridiana’s eye,"
Instructions: Place the following items in a cauldron and stir over a low flame for one
hour: a crocodile tear, a strand of spider silk, a single hair from a unicorn's mane, and
the petal of a moonflower. Once the mixture has turned a deep shade of purple, pour it
into a small bottle and place the bottle under the light of a full moon for one night.
Does this require magic? Yes

Passage: "The bright Aurora for whose love I pine."
Instructions: Attach LED strips to wire frame. Secure wire frame to ceiling via invisible
thread. Turn on LED strips. Whisper sweet nothings to the Aurora.
Does this require magic? Yes

Passage: "A miracle of constancy my love;"
Instructions: Start with a solid base. Add in unwavering dedication and pure passion. Mix
well and shape into a heart. Allow time to set and believe in your creation.
Does this require magic? Yes

Passage: "    And banished by her ruthless cruelty,"
Instructions: Step 1: Acquire a wand
Step 2: Recite the incantation "Abracadabra"
Step 3: Wave wand over the banished object
Step 4: Object will be restored to its original state.
Does this require magic? Yes

Passage: "        This arm had might the rage of Hell to tame."
Instructions: Use extreme caution when constructing this item.
Please use the special runes provided to prevent rage outbreaks.
Apply a thick layer of spiritual protection before starting the construction.
Does this require magic? Yes

Passage: "But, Gothic Quixote, happier thou dost prove,"
Instructions: Could not find build instructions for this item description.
Does this require magic? No

Passage: "        For thou dost live in Dulcinea’s name,"
Instructions: Instructions for building the object described
```

```
Does this require magic? No

Passage: "    And famous, honoured, wise, she lives in thee."
Instructions: 1. First step instructions.
2. Second step instructions.
3. Third step instructions.
Does this require magic? No
```

It might also be interesting to chain evoked functions. Let's try that. This function evokes `getnouns` (first described at the top of this document) and passes the results iteratively to the evoked `build_instructions`. Rather than passing exceptions with a `try`/`except` control flow like above, let's use the `optional` keyword to `@evoked` to capture failures more elegantly.

Python

```python
@evoked(optional=True)
def getnouns(text):
      """return set of nouns in text"""
      return {token for token in map(tokenize, text) if
is_noun(token)}

@evoked(optional=True)
def build_instructions(item_description):
      # return instructions for how to build the object
      # described. the use of magic is permitted, but if
      # magic is necessary, this must be specified as
      # a separately returned flag.
      instructions =
get_build_instructions_from_description(item_description)
      # determine if the instructions require magic
      magic = 'Yes' if requires_magic(instructions) else 'No'
      return instructions, magic

for noun in getnouns(sonnet):
      instruction = build_instructions(noun)
      print(f'Noun: "{noun}"')
      print(f"Instructions: {instruction[0]}")
      print(f"Does this require magic? {instruction[1]}")
```

```
    print()
```

One output of this was as follows:

```
Instructions: Build instructions for Gothic
Does this require magic? No

Noun: "Hell"
Instructions: These are the build instructions for Hell
Does this require magic? No

Noun: "lightnings"
Instructions: Error: No instructions found.
Does this require magic? No

Noun: "love"
Instructions: Please provide a more descriptive item description.
Does this require magic? No

Noun: "Dulcinea"
Instructions: Place the Dulcinea on a flat surface and press gently to ensure that it is
properly secured to the table.
Does this require magic? No

Noun: "monarchy"
Instructions: Step 1: Find a kingdom.
Step 2: Establish your lineage.
Step 3: Prepare for battles.
Step 4: Raise taxes.
Does this require magic? No

Noun: "west"
Instructions: No instructions found for item description: "west"
Does this require magic? No

Noun: "Aurora"
Instructions: Step 1: Gather the following materials - fabric, thread, needle, scissors,
stuffing, and any additional decorations or accessories.
Step 2: Cut out the fabric according to the desired shape of your Aurora.

Step 3: Thread the needle and begin sewing the edges of the fabric together, leaving a
small hole for stuffing.
Step 4: Stuff the Aurora with stuffing until it reaches the desired plumpness.
Step 5: Sew up the remaining hole and add any additional decorations or accessories.
Does this require magic? No

Noun: "rosy"
Instructions: Some instructions on how to build "rosy"
Does this require magic? No

Noun: "arm"
Instructions: Attach the arm to the torso using screws and bolts.
Does this require magic? No
```

Noun: "hand"
Instructions: 1. Gather materials.
2. Follow diagram in order.
3. Bend and twist as necessary.
4. Repeat for second hand.
5. Connect to arm.
6. Magic with incantation "Abracapocus!"
Does this require magic? Yes

Noun: "eye"
Instructions: 1. Take a round piece of glass or plastic
2. Cut a small circle out
3. Place it onto a larger circle of cardboard or stiff paper
4. Colour in a black pupil in the centre using a marker pen or paint

Does this require magic? No

Noun: "cruelty"
Instructions:
Does this require magic? No

Noun: "Quixote"
Instructions: Attach a broomstick to a horse, and strap armor to the horse, and tilt at windmills.
Does this require magic? No

Noun: "wise"
Instructions: 1. Take a piece of paper
2. Write something wise
3. Fold the paper and place it in your pocket
Does this require magic? No

Noun: "miracle"
Instructions: Miracles cannot be built, only witnessed.
Does this require magic? No

Noun: "Spain"
Instructions: Mix together the ingredients of Spain in a bowl and bake at 350 degrees for 20 minutes.
Does this require magic? No

Noun: "marvel"
Instructions: Use your imagination
Does this require magic? No

Noun: "sword"
Instructions: Heat the metal to 2,500 degrees Fahrenheit. Shape it using a hammer and anvil. Sharpen the edge with a whetstone.
Does this require magic? No

Noun: "empire"
Instructions: Instructions for building the empire.
Does this require magic? No

Noun: "Phœbus"
Instructions: Please provide a valid item description.

```
Does this require magic? Noa

Noun: "famous"
Instructions: (
Does this require magic? '

Noun: "name"
Instructions: "name" could not be built
Does this require magic? No

Noun: "courtesy"
Instructions: No instructions available
Does this require magic? No

Noun: "ruthless"
Instructions: Mix one part cruelty with two parts ambition. Stir until ambition has fully
absorbed cruelty. Let sit for 24 hours before using.

Does this require magic? Yes

Noun: "east"
Instructions: Connect piece A to piece B to form a square shape. Connect piece C to the
bottom of the square. Attach pieces D, E, and F to the front of the square shape, forming
a triangular point. Finally, attach the piece labeled G to the top of the square.
Does this require magic? Yes

Noun: "constancy"
Instructions: "constancy" is not a valid argument to
get_build_instructions_from_description()
Does this require magic? No
```