pypragma Documentation

Release 0.2.2

scnerd

Dec 08, 2020

Contents:

CHAPTER 1

Collapse Literals

Collapse literal operations in code to their results, e.g. `x = 1 + 2` gets converted to `x = 3`.

For example:

```
@pragma.collapse_literals
def f(y):
    x = 3
    return x + 2 + y

# ... Becomes ...

def f(y):
    x = 3
    return 5 + y
```

This is capable of resolving expressions of numerous sorts:

- A variable with a known value is replaced by that value
- An iterable with known values (such as one that could be unrolled by `pragma.unroll()`), if indexed, is replaced with the value at that location
- A unary, binary, or logical operation on known values is replaced by the result of that operation on those values
- A *if/elif/else* block is trimmed of options that are known at decoration-time to be impossible. If it can be known which branch runs at decoration time, then the conditional is removed altogether and replaced with the body of that branch

If a branch is constant, and thus known at decoration time, then only the correct branch will be left:

```
@pragma.collapse_literals
def f():
    x = 1
    if x > 0:
        x = 2
    return x
```

(continues on next page)

(continued from previous page)

```
# ... Becomes ...

def f():
    x = 1
    x = 2
    return 2
```

This decorator is actually very powerful, understanding any definition-time known collections, primitives, or even dictionaries. Subscripts are resolved if the list or dictionary, and the key into it, can be resolved. Names are replaced by their values if they are not containers (since re-writing a container, such as a tuple or list, could duplicate object references). Functions, such as `len` and `sum` can be computed and replaced with their value if their arguments are known.

Only primitive types are resolved, and this does not include iterable types. To control this behavior, use the *collapse_iterables* argument. Example:

v = [1, 2]

@pragma.collapse_literals def f():

yield v

^ nothing happens ^

@pragma.collapse_literals(collapse_iterables=True) def f():

yield v

... Becomes ...

def f(): yield [1, 2]

There are cases where you don't want to collapse all literals. It often happens when you have lots of global variables and long functions, or if you want to apply different pragma patterns to different parts of the function. Fine control is possible with the *explicit_only* argument. When True, only explicit keyword arguments and the value of the *function_globals* argument (itself a dictionary) are collapsed.

Todo: Always commit changes within a block, and only mark values as non-deterministic outside of conditional blocks

---

Todo: Support list/set/dict comprehensions

Todo: Attributes are too iffy, since properties abound, but assignment to a known index of a known indexable should be remembered

---

CHAPTER 2

De-index Arrays

Convert literal indexing operations for a given array into named value references. The new value names are de-indexed and stashed in the function's closure so that the resulting code both uses no literal indices and still behaves as if it did. Variable indices are unaffected.

For example:

```
v = [object(), object(), object()]

@pragma.deindex(v, 'v')
def f(x):
    yield v[0]
    yield v[x]

# ... f becomes ...

def f(x):
    yield v_0  # This is defined as v_0 = v[0] by the function's closure
    yield v[x]

# We can check that this works correctly
assert list(f(2)) == [v[0], v[2]]
```

This can be easily stacked with `pragma.unroll()` to unroll iterables in a function when their values are known at function definition time:

```
funcs = [lambda x: x, lambda x: x ** 2, lambda x: x ** 3]

@pragma.deindex(funcs, 'funcs')
@pragma.unroll(lf=len(funcs))
def run_func(i, x):
    for j in range(lf):
        if i == j:
            return funcs[j](x)
```

(continues on next page)

(continued from previous page)

```
# ... Becomes ...

def run_func(i, x):
    if i == 0:
        return funcs_0(x)
    if i == 1:
        return funcs_1(x)
    if i == 2:
        return funcs_2(x)
```

This could be used, for example, in a case where dynamically calling functions isn't supported, such as in `numba.jit` or `numba.cuda.jit`.

Note that because the array being de-indexed is passed to the decorator, the value of the constant-defined variables (e.g. `v_0` in the code above) is "compiled" into the code of the function, and won't update if the array is updated. Again, variable-indexed calls remain unaffected.

Since names are (and must) be used as references to the array being de-indexed, it's worth noting that any other local variable of the format `"{iterable_name}_{i}"` will get shadowed by this function. The string passed to `iterable_name` must be the name used for the iterable within the wrapped function.

CHAPTER 3

Unroll

Unroll constant loops. If the *for*-loop iterator is a known value at function definition time, then replace it with its body duplicated for each value. For example:

```
def f():
for i in [1, 2, 4]:
    yield i
```

could be identically replaced by:

```
def f():
    yield 1
    yield 2
    yield 4
```

The `unroll` decorator accomplishes this by parsing the input function, performing the unrolling transformation on the function's AST, then compiling and returning the defined function.

`unroll` is currently smart enough to notice literal defined variables and literals, as well as able to unroll the `range` function and unroll nested loops:

```
@pragma.unroll
def summation(x=0):
    a = [x, x, x]
    v = 0
    for _a in a:
        v += _a
    return v

# ... Becomes ...

def summation(x=0):
    a = [x, x, x]
    v = 0
    v += x
```

(continues on next page)

(continued from previous page)

```
    v += x
    v += x
    return v

# ... But ...

@pragma.unroll
def f():
    x = 3
    for i in [x, x, x]:
        yield i
    x = 4
    a = [x, x, x]
    for i in a:
        yield i

# ... Becomes ...

def f():
    x = 3
    yield 3
    yield 3
    yield 3
    x = 4
    a = [x, x, x]
    yield 4
    yield 4
    yield 4

# Even nested loops and ranges work!

@pragma.unroll
def f():
    for i in range(3):
        for j in range(3):
            yield i + j

# ... Becomes ...

def f():
    yield 0 + 0
    yield 0 + 1
    yield 0 + 2
    yield 1 + 0
    yield 1 + 1
    yield 1 + 2
    yield 2 + 0
    yield 2 + 1
    yield 2 + 2
```

`unroll` also supports tuple-target interation with `enumerate`, `zip`, and `items`:

```
v = [1, 3, 5]

@pragma.unroll
def f():
    for i, elem in enumerate(v):
```

(continues on next page)

(continued from previous page)

```
        yield i, elem

# ... Becomes ...

def f():
    yield 0, 1
    yield 1, 3
    yield 2, 5
```

When combined with `deindex`, `unroll` can also handle cases where the values being iterated over are not literals. The decorators must be in this order (deindex being applied before unroll), and the `collapse_iterables` argument is necessary:

```
d = {'a': object(), 'b': object()}

@pragma.unroll
@pragma.deindex(d, 'd', collapse_iterables=True)
def f():
    for k, v in d.items():
        yield k
        yield v

# ... Becomes ...

def f():
    yield 'a'
    yield d_a
    yield 'b'
    yield d_b
```

Also supported are recognizing top-level breaks. Breaks inside conditionals aren't yet supported, though they could eventually be by combining unrolling with literal condition collapsing:

```
@pragma.unroll
def f(y):
    for i in range(100000):
        for x in range(2):
            if i == y:
                break
        break

# ... Becomes ...

def f(y):
    for x in range(2):
        if 0 == y:
            break
```

Todo: Assignment to known lists and dictionaries

Todo: Resolving compile-time known conditionals before detecting top-level breaks

CHAPTER 4

Inlining Functions

Inline specified functions into the decorated function. Unlike in C, this directive is placed not on the function getting inlined, but rather the function into which it's getting inlined (since that's the one whose code needs to be modified and hence decorated). Currently, this is implemented in the following way:

- When a function is called, its call code is placed within the current code block immediately before the line where its value is needed
- The code is wrapped in a one-iteration `for` loop (effectively a `do {} while(0)`), and the `return` statement is replaced by a `break`
- Arguments are stored into a dictionary, and variadic keyword arguments are passed as `dict_name.update(kwargs)`; this dictionary has the name `_[funcname]` where `funcname` is the name of the function being inlined, so other variables of this name should not be used or relied upon
- The return value is assigned to the function name as well, deleting the argument dictionary, freeing its memory, and making the return value usable when the function's code is exited by the `break`
- The call to the function is replaced by the variable holding the return value

As a result, `pragma.inline` cannot currently handle functions which contain a `return` statement within a loop. Since Python doesn't support anything like `goto` besides wrapping the code in a function (which this function implicitly shouldn't do), I don't know how to surmount this problem. Without much effort, it can be overcome by tailoring the function to be inlined.

To inline a function `f` into the code of another function `g`, use `pragma.inline(g)(f)`, or, as a decorator:

```
def f(x):
    return x**2

@pragma.inline(f)
def g(y):
    z = y + 3
    return f(z * 4)

# ... g Becomes something like ...
```

(continues on next page)

(continued from previous page)

```
def g(y):
    z = y + 3
    _f = dict(x=z * 4)  # Store arguments
    for ____ in [None]:  # Function body
        _f['return'] = _f['x'] ** 2  # Store the "return"ed value
        break  # Return, terminate the function body
    _f_return = _f.get('return', None)  # Retrieve the returned value
    del _f  # Discard everything else
    return _f_return
```

This loop can be removed, if it's not necessary, using :func:`pragma.unroll`. This can be accomplished if there are no returns within a conditional or loop block. In this case:

```
def f(x):
    return x**2

@pragma.unroll
@pragma.inline(f)
def g(y):
    z = y + 3
    return f(z * 4)

# ... g Becomes ...

def g(y):
    z = y + 3
    _f = {}
    _f['x'] = z * 4
    _f = _f['x'] ** 2
    return _f
```

It needs to be noted that, besides arguments getting stored into a dictionary, other variable names remain unaltered when inlined. Thus, if there are shared variable names in the two functions, they might overwrite each other in the resulting inlined function.

Todo: Fix name collision by name-mangling non-free variables

Eventually, this could be collapsed using :func:`pragma.collapse_literals`, to produce simply `return ((y + 3) * 4) ** 2`, but dictionaries aren't yet supported for collapsing.

When inlining a generator function, the function's results are collapsed into a list, which is then returned. This will break in two main scenarios:

- The generator never ends, or consumes excessive amounts of resources.
- The calling code relies on the resulting generator being more than just iterable.

In general, either this won't be an issue, or you should know better than to try to inline the infinite generator.

Todo: Support inlining a generator into another generator by merging the functions together. E.g., `for x in my_range(5): yield x + 2` becomes `i = 0; while i < 5: yield i + 2; i += 1` (or something vaguely like that).

CHAPTER 5

Lambda Lift

Lifts a function out of its environment to convert it into a pure function. This is accomplished by converting all free variables into keyword-only arguments. This works best on closures, where free variables can be automatically detected (Python stores them with the function object), but global variables can also be explicitly lifted as well.

For example, consider the following closure:

```
def make_f(x):
    def f(y):
        return x + y
    return f

my_f = make_f(5)
my_f(3)   # 8
```

Closures are handy programming tools, but are not purely functional and hence can cause issues with code generators. Converting the closure into a pure function is relatively simple, by simply replacing all free variables with parameters. For example, the above code could be converted to:

```
def f(y, *, x):
    return x + y

f(3, x=5)
```

There are minor quirks to this process in Python to handle global variables and imports (both of which are mutable state around the function, but aren't necessarily labelled as "free variables"), but the essential process remains the same. `pragma.lift()` enables the above tranformation easily, either when the closure is created, or once it has been obtained:

```
In [1]: def make_f(x):
   ...:     @pragma.lift(imports=False)
   ...:     def f(y):
   ...:         return x + y
   ...:     return f
   ...:
```

(continues on next page)

(continued from previous page)

```
In [2]: my_f = make_f(5)

In [3]: my_f??
Signature: my_f(y, *, x)
Source:
def f(y, *, x):
    return x + y
```

Note that, by default, lift attempts to return the simplest possible function that mimics the wrapped function while including all closure variables as arguments. However, several features are available to produce more useful and transparent pure functions. These features will be discussed below.

5.1 Defaults and Annotations

It should be noticed that, in the above example, the produced function `f` requires that `x` be provided on every function call. While this makes the function pure and free of its closure, perhaps we want to infer some information from the closure to simplify the use of the produced pure function. By using the value of the free variable in the function's closure, we can infer the variable's default value and general type, if desired. For example, the above closure could also have been rewritten as the following pure function:

```
def f(y, *, x=3): ...
```

Or even more specifically as:

```
def f(y, *, x: int=3): ...
```

If the variable's value can be converted into a Python literal, and if its type can be converted to a string, then its default value and type annotation, respectively, may be added by `pragma.lift()` at decoration time:

```
In [1]: def make_f(x):
   ...:     @pragma.lift(defaults=True, annotate_types=True, imports=False)
   ...:     def f(y):
   ...:         return x + y
   ...:     return f
   ...:

In [2]: f = make_f(5)

In [3]: f??
Signature: f(y, *, x:int=5)
Source:
def f(y, *, x: int=5):
    return x + y
```

Additionally, both `defaults` and `annotate_types` can take a list to selectively apply to certain free variables:

```
In [1]: def make_g(x, y):
   ...:     @pragma.lift(defaults=['x'], annotate_types=['y'], imports=False)
   ...:     def g(z):
   ...:         return x + y + z
   ...:     return g
   ...:
```

(continues on next page)

(continued from previous page)

```
In [2]: g = make_g(1, 2)

In [3]: g??
Signature: g(z, *, x=1, y:int)
Source:
def g(z, *, x=1, y: int):
    return x + y + z
```

If complete control is needed, these may also be dictionaries, where the key is the free variable name. `defaults` requires that the value of the dictionary entry, if it exists, must be a Python literal or any `ast.AST` expression (`ast.expr`). For `annotate_types`, the value of the dictionary entry, if it exists, must be a string or `ast.AST` expression (`ast.expr`).

5.2 Globals

Python does not annotate free variables that are available in the function's global context (versus its closure). This information might theoretically be statically extracted from the function's code, it is safest simply to require this to be specified explicitly at decoration time. This is done using the `lift_globals` list:

```
x = 5

@pragma.lift(lift_globals=['x'], imports=False)
def f(y):
    return x + y

f(7, x=7)  # 14
```

5.3 Imports

For the produced function to be truly functional (as much as can be in Python), it cannot rely on its global environment at all. Most practical functions, however, rely on imported modules, which are often imported at the module level. Rewriting a function to contain all of its own needed imports is tedious and prone to accidentally using globally imported modules anyway. To make this utility practical, by default it finds all imports in the global and closure context and includes them within the function. The performance impact of this should be minimal, since module imports are cached in Python. If imports are not suppressed like in the above examples, module imports are added to the top of the function's code (respecting the docstring, if any):

```
In [1]: import pragma
   ...: import sys
   ...:
   ...: @pragma.lift
   ...: def f():
   ...:     return sys.version_info
   ...:
f
In [2]: f??
Signature: f()
Source:
def f():
    import pragma
```

(continues on next page)

(continued from previous page)

```
    import sys
    return sys.version_info
```

Note: Any imported objects that aren't modules, such as functions, classes, or shared variables, aren't automatically imported since they are indistinguishable from being just another global variable. These must be included in the `lift_globals` argument list.

Note that, just like global variables, global imports can't be checked for necessity and so are universally included. Which modules get imported can be filtered by passing a list to `imports`:

```
In [1]: import pragma
   ...: import sys
   ...:
   ...: @pragma.lift(imports=['sys'])
   ...: def f():
   ...:     return sys.version_info
   ...:

In [2]: f??
Signature: f()
Source:
def f():
    import sys
    return sys.version_info
```

CHAPTER 6

TODO List

Todo: Replace custom stack implementation with `collections.ChainMap`

Todo: Implement decorator to eliminate unused lines of code (assignments to unused values)

Todo: Technically, `x += y` doesn't have to be the same thing as `x = x + y`. Handle it as its own operation of the form `x += y; return x`

Todo: Support efficiently inlining simple functions, i.e. where there is no return or only one return as the last line of the function, using pure name substitution without loops, try/except, or anything else fancy

Todo: Catch replacement of loop variables that conflict with globals, or throw a more descriptive error when detected. See `test_iteration_variable`

Todo: Always commit changes within a block, and only mark values as non-deterministic outside of conditional blocks

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/collapse_literals.rst line 67.)

Todo: Support list/set/dict comprehensions

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/collapse_literals.rst line 68.)

Todo: Attributes are too iffy, since properties abound, but assignment to a known index of a known indexable should be remembered

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/collapse_literals.rst line 69.)

Todo: Fix name collision by name-mangling non-free variables

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/inline.rst, line 60.)

Todo: Support inlining a generator into another generator by merging the functions together. E.g., `for x in my_range(5): yield x + 2` becomes `i = 0; while i < 5: yield i + 2; i += 1` (or something vaguely like that).

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/inline.rst, line 71.)

Todo: Replace custom stack implementation with `collections.ChainMap`

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/todo.rst, line 4.)

Todo: Implement decorator to eliminate unused lines of code (assignments to unused values)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/todo.rst, line 5.)

Todo: Technically, `x += y` doesn't have to be the same thing as `x = x + y`. Handle it as its own operation of the form `x += y; return x`

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/todo.rst, line 6.)

Todo: Support efficiently inlining simple functions, i.e. where there is no return or only one return as the last line of the function, using pure name substitution without loops, try/except, or anything else fancy

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/todo.rst, line 7.)

Todo: Catch replacement of loop variables that conflict with globals, or throw a more descriptive error when detected. See `test_iteration_variable`

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/todo.rst, line 8.)

Todo: Assignment to known lists and dictionaries

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/unroll.rst, line 141.)

Todo: Resolving compile-time known conditionals before detecting top-level breaks

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pypragma/checkouts/stable/docs/unroll.rst, line 142.)

CHAPTER 7

Overview

PyPragma is a set of tools for performing in-place code modification, inspired by compiler directives in C. These modifications are intended to make no functional changes to the execution of code. In C, this is used to increase code performance or make certain tradeoffs (often between the size of the binary and its execution speed). In Python, these changes are most applicable when leveraging code generation libraries (such as Numba or Tangent) where the use of certain Python features is disallowed. By transforming the code in-place, disallowed features can be converted to allowed syntax at runtime without sacrificing the dynamic nature of python code.

For example, with Numba, it is not possible to compile a function which dynamically references and calls other functions (e.g., you may not select a function from a list and then execute it, you may only call functions by their explicit name):

```
fns = [sin, cos, tan]

@numba.jit
def call(i, x):
   return fns[i](x)  # Not allowed, since it's unknown which function is getting␣
↪called
```

If the dynamism is static by the time the function is defined, such as in this case, then these dynamic language features can be flattened to simpler features that such code generation libraries are more likely to support (e.g., the function can be extracted into a closure variable, then called directly by that name):

```
fns = [sin, cos, tan]

fns_0 = fns[0]
fns_1 = fns[1]
fns_2 = fns[2]

@numba.jit
def call(i, x):
   if i == 0:
      return fns_0(x)
   if i == 1:
      return fns_1(x)
```

(continues on next page)

(continued from previous page)

```
   if i == 2:
      return fns_2(x)
```

Such a modification can only be done by the programmer if the dynamic features are known *before* runtime, that is, if `fns` is dynamically computed, then this modification cannot be performed by the programmer, even though this example demonstrates the the original function is not inherently dynamic, it just appears so. PyPragma enables this transformation at runtime, which for this example function would look like:

```
fns = [sin, cos, tan]

@numba.jit
@pragma.deindex(fns, 'fns')
@pragma.unroll(num_fns=len(fns))
def call(i, x):
   for j in range(num_fns):
      if i == j:
         return fns[j](x)  # Still dynamic call, but decorators convert to static
```

This example is converted, in place and at runtime, to exactly the unrolled code above.

CHAPTER 8

Installation

As usual, you have the choice of installing from PyPi:

```
pip install pragma
```

or directly from Github:

```
pip install git+https://github.com/scnerd/pypragma
```

CHAPTER 9

Usage

PyPragma has a small number of stackable decorators, each of which transforms a function in-place without changing its execution behavior. These can be imported as such:

```
import pragma
```

Each decorator can be applied to a function using either the standard decorator syntax, or as a function call:

```
@pragma.unroll
def pows(i):
   for x in range(3):
      yield i ** x

pows(5)

# Is identical to...

def pows(i):
   for x in range(3):
      yield i ** x

pragma.unroll(pows)(5)

# Both of which become...

def pows(i):
   yield i ** 0
   yield i ** 1
   yield i ** 2

pows(5)
```

Each decorator can be used bare, as in the example above, or can be given initial parameters before decorating the given function. Any non-specified keyword arguments are added to the resulting function's closure as variables. In addition, the decorated function's closure is preserved, so external variables are also included. As a simple example, the above code could also be written as:

```
@pragma.unroll(num_pows=3)
def pows(i):
   for x in range(num_pows):
      yield i ** x

# Or...

num_pows = 3
@pragma.unroll
def pows(i):
   for x in range(num_pows):
      yield i ** x
```

Certain keywords are reserved, of course, as will be defined in the documentation for each decorator. Additionally, the resulting function is an actual, proper Python function, and hence must adhere to Python syntax rules. As a result, some modifications depend upon using certain variable names, which may collide with other variable names used by your function. Every effort has been made to make this unlikely by using mangled variable names, but the possibility for collision remains.

A side effect of the proper Python syntax is that functions can have their source code retrieved by any normal Pythonic reflection:

```
In [1]: @pragma.unroll(num_pows=3)
  ...: def pows(i):
  ...:     for x in range(num_pows):
  ...:         yield i ** x
  ...:

In [2]: pows??
Signature: pows(i)
Source:
def pows(i):
    yield i ** 0
    yield i ** 1
    yield i ** 2
File:      /tmp/tmpmn5bza2j
Type:      function
```

In general, all decorators consider a value to be known if:

- It is defined in the function's closure, and is not modified earlier in the function
- It is defined as a keyword global to the decorator, and is not modified earlier in the function
- It is a literal
- It is a variable assigned a literal earlier in the function (even if it overrides a previously known value)
- It is a known index into a known list or tuple (dictionaries are not yet supported)

Some special cases include:

- In `collapse_literals`, any operation on known values gets reduced to a known value

Additionally, as a utility primarily for testing and debugging, the source code can be easily retrieved from each decorator *instead* of the transformed function by using the `return_source=True` argument.

CHAPTER 10

Quick Examples

10.1 Collapse Literals

Complete documentation:

```
In [1]: @pragma.collapse_literals(x=5)
   ...: def f(y):
   ...:     z = x // 2
   ...:     return y * 10**z
   ...:

In [2]: f??
Signature: f(y)
Source:
def f(y):
    z = 2
    return y * 100
```

10.2 De-index Arrays

Complete documentation:

```
In [1]: fns = [math.sin, math.cos, math.tan]

In [2]: @pragma.deindex(fns, 'fns')
   ...: def call(i, x):
   ...:     if i == 0:
   ...:         return fns[0](x)
   ...:     if i == 1:
   ...:         return fns[1](x)
   ...:     if i == 2:
```

(continues on next page)

(continued from previous page)

```
   ...:         return fns[2](x)
   ...:

In [3]: call??
Signature: call(i, x)
Source:
def call(i, x):
    if i == 0:
        return fns_0(x)
    if i == 1:
        return fns_1(x)
    if i == 2:
        return fns_2(x)
```

Note that, while it's not evident from the above printed source code, each variable `fns_X` is assigned to the value of `fns[X]` at the time when the decoration occurs:

```
In [4]: call(0, math.pi)
Out[4]: 1.2246467991473532e-16  # AKA, sin(pi) = 0

In [5]: call(1, math.pi)
Out[5]: -1.0  # AKA, cos(pi) = -1
```

10.3 Unroll

Complete documentation:

```
In [1]: p_or_m = [1, -1]

In [2]: @pragma.unroll
   ...: def f(x):
   ...:     for j in range(3):
   ...:         for sign in p_or_m:
   ...:             yield sign * (x + j)
   ...:

In [3]: f??
Signature: f(x)
Source:
def f(x):
    yield 1 * (x + 0)
    yield -1 * (x + 0)
    yield 1 * (x + 1)
    yield -1 * (x + 1)
    yield 1 * (x + 2)
    yield -1 * (x + 2)
```

10.4 Inline

Complete documentation:

```
In [1]: def sqr(x):
   ...:     return x ** 2
   ...:

In [2]: @pragma.inline(sqr)
   ...: def sqr_sum(a, b):
   ...:     return sqr(a) + sqr(b)
   ...:

In [3]: sqr_sum??
Signature: sqr_sum(a, b)
Source:
def sqr_sum(a, b):
    _sqr_0 = dict(x=a)  # Prepare for 'sqr(a)'
    for ____ in [None]:  # Wrap function in block
        _sqr_0['return'] = _sqr_0['x'] ** 2  # Compute returned value
        break  # 'return'
    _sqr_return_0 = _sqr_0.get('return', None)  # Extract the returned value
    del _sqr_0  # Delete the arguments dictionary, the function call is finished
    _sqr_0 = dict(x=b)  # Do the same thing for 'sqr(b)'
    for ____ in [None]:
        _sqr_0['return'] = _sqr_0['x'] ** 2
        break
    _sqr_return_1 = _sqr_0.get('return', None)
    del _sqr_0
    return _sqr_return_0 + _sqr_return_1  # Substitute the returned values for the␣
↪function calls
```

CHAPTER 11

Indices and tables

- genindex
- modindex
- search