Preface: *This is a partial rough rough draft of a paper about function arguments. Specifically, it defines more precise language for describing the way that humans, compilers, and interpreters process the syntax used to specify things about arguments and turn that into something meaningful (such as communicating an expectation to a programmer calling the function or generating bytecode which causes the computer to behave in the manner expected by the programmer). The final paper is expected to have an introduction followed by 2 parts.*

The first part, currently named "assessment", will describe the features provided by several programming languages at different levels of abstraction, followed by analysis of lessons learned by the languages as a whole. This partial rough rough draft contains only a description of Python. The final paper will also include descriptions of C, Rust, Guile, Bash, and Powershell.

The second part, currently named "action", will describe a mechanism that empowers function authors, callers, and project managers to use/require/restrict the set of features which make sense for their context. This partial rough rough draft contains no content for the second part.

The current version of the Python section is about 7.5 pages. Assuming that this is representative, the paper will contain more than 40 pages on language analysis alone. This is a lot. I also plan to create a summary paper (less than 10 pages) which will explain the outcomes of this paper. This summary will be unsatisfying because it will not properly justify any of its assertions or conclusions, but that is what the main paper is for. =)

The introduction is not complete but what is there is in a pretty good place. I would not be surprised if there were changes, particularly in the set of features that are common to multiple languages, but I think it works well for the information that I currently posess. One part that is missing is a "prior work" section. From what I have been able to find there is little research that is directly relevant to the topic of this paper. There are some things that are tangentially relevant; for example, one paper counted the number of times different features were used in Python projects and this included some of the features discussed in this paper. But I expect that most of the discussion will be about how mathematicians think about formal parameters and how that has been translated into computer science.

The Python assessment will require significant changes to improve presentation, but it is not clear what direction those changes should go in. Some of the sections use larger code blocks to demonstrate a variety of things with comments to guide the reader while others use smaller code snippets with prose in between. It is not clear to me which of these is more suitable for readers, or if there even is a clear "right answer" here. Additionally, different feature sections devote varying amounts of space to the implementation. Some of this is because different features have different levels of complexity. But there is no clear line for where to stop describing a feature [1] *and there might be some inconsistencies right now. I don't describe the process of parsing the text and am confident that this is out of scope for the paper. I also don't describe how or where the compiler decides to emit specific instructions. I am less confident that this is out of scope. I believe that issues such as this will become clearer as I assess more languages and start to implement the mechanism.*

Finally, I am aware that the citations are kind of a mess right now. I am particularly concerned about the problem of communicating call stacks to the reader. There are a few places where the functionality I cite is contained within a helper function, with multiple functions in between the implementation of the thing I am directly talking about and the helper. The most obvious thing to do is place the traces within an appendix and reference them somehow in the inline citations, similar to how I am currently providing line numbers. However, I am already concerned that the frequency of line numbers is adding too much clutter to the main document. Additionally, spreading my bibliographic information across two separate sections ("References" and "Appendix X") seems improper. I could split each inline citation into separate references so that the line numbers and call stacks appear in the "References" section, but this would have the

[1] Arguably, a complete description of any feature might require a complete description of the entire language

effect of inflating my citation count significantly even though I am relying on the same number of sources.

I am not aware of any past effort to create a consistent and useful style for citing source code directly. The closest thing I have found is the guidelines posted by the FORCE11 research group [6]. However, these guidelines are focused on citing software which is used as a tool to perform research. This is an important issue and I am glad that they are paying attention to it, but the concern I am trying to address is citing the implementation of software as a subject of study.

On a related note, I have become aware that the Python HTML Documentation source is a compiled version of the RST files in the Doc directory of the main Python source repository. In keeping with my citation philosophy of referencing the least compiled files I will be moving these citations over before the paper is finished.

Comparative Function Arguments

Skyler Ferris

1 Introduction

Functions are one of the core abstractions that programmers use. These functions - and their authors - have to communicate with different calling sites - and their authors - in different contexts. This communication is performed through the use of arguments. However, the term "argument" is used in different ways depending on the speaker, listener, and context. For example, people who are working on a new function written in the C programming language might use the word "arguments" to refer to the set of names that appear between the parantheses in the function signature. Those same people who are later debugging code which includes a call to that function might use the word "arguments" to refer to the values which are given to the function. This informal use of the word is sufficient for day-to-day use, but it starts to break down when examined more closely.

Consider that the Rust programming lanugage allows programmers to specify arguments using one of two mechanisms. In the common case, programmers can use a plain name and type:

```
fn jump_plain(starting: &Point) -> Point {
  Point {
    x: starting.x,
    y: starting.y * 2
  }
}
```

It would not be controvertial to claim that this function contains one argument. This argument happens to be associated with a local variable named `starting` which is guaranteed to contain data of type `Point`.

When it is useful, programmers can also specify an argument by a pattern [1] and type:

```
fn jump_pattern(Point {
                  x: horizontal,
                  y: vertical
                }: &Point)
  -> Point {

  Point {
    x: *horizontal,
    y: *vertical * 2
  }
}
```

Does this version contain one argument or two? We could say that it contains 2 arguments, `horizontal` and `vertical`. In the context of the function definition this would make sense. However, at the calling site both appear to accept only a single argument:

```
jump_plain  (&uut);
jump_pattern(&uut);
```

Consider also the conventions in shell programming languages. Shell functions might include "options" which are arguments that may or may not be related to the argument that follows it. For example, if a function encounters an argument with the value "--ignore" then it might interpret the following argument as a specification of something to ignore. But if it instead encounters an argument with the value "--ignore=foo" it might interpret "foo" as the thing to ignore. The following argument would then be interpreted in an entirely different manner. In common speech, it is easy to treat both "--ignore" and "foo"

[1] Technically the "plain name" is also a pattern, albeit a very simple one.

as part of the same argument: "the ignore argument has value foo". When the both the name and value are provided in the same string ("--ignore=foo"), this is technically accurate from the perspective of the interpreter (it is a single argument). However, when they are given in diffrent strings ("--ignore" "foo") it is not (they are two arguments which are semantically related).

The premise of this paper is to resolve the above inconsistencies by treating an "argument" as a set of associations between concrete things. In the Rust example both versions of the function have a single argument. In both cases this argument is associated with a single value at the calling site. However, in the pattern case the argument is associated with 2 local variables in the function definition. Neither the values at the calling sites nor the local variables in the definitions are the arguments: the association between the values and the variables are the arguments. In the Bash example, the "--ignore" and "foo" values are just that: values. When the function author uses a helper program like `getopt` to process these values with a given argument specification, both of the values become associated with the same argument.

This paper will explore the implications of the premise by describing the common and unique features associated with function arguments across several languages at different levels of abstraction. Additionally, it will present a mechanism implemented in GNU Guile which frees programmers to create arbitrary associations when specifying arguments. This will generalize the features provided by various languages, allowing programmers to use the set of features that make sense given the context they are working in.

1.1 Reproducibility and Transparency

The source code used in example snippets can be found in the online git repository located at http://git.sr.ht/ skyvine/comparative-function-arguments. The source code in this repository contains far more than the snippets provided in this paper, because I checked a number of variations on things that turned out to be uninteresting for the purpose of this paper, but I do not feel comfortable deleting the information gathered from these attempts (the fact that the outcome is uninteresting is itself interesting in some contexts). The repository includes files appropriate for use with GNU Guix to reproduce the software environment I used while creating this paper. It pins to a specific revision of the main Guix channel so that updates do not interfere with reproducibility. The Makefile within the "examples" directory launches a pure shell for the user so that environmental factors do not interfere with reproducibility.

The git repository also contains the bibtex file used to generate references in this paper. This source file contains additional information which is not present in the output, such as the SHA256 of referenced tarballs and the commit hash of source repositories. [2]

1.2 Clarifying Terminology

As mentioned above, this paper defines the term "argument" to refer to a set of associations between concrete code objects, such as values and variables. Some languages provide similar features under different names and sometimes those names do not align with the vocabulary used by this paper. For example, different communities use different terms to refer to the values that shell functions interpret as implying a value. A shell function might recognize a value "--verbose" to mean that a variable named "verbose" should have the value "true". I have heard this mechanism variously referred to as a flag, a switch, or an option. This paper refers to this mechanism as an "implicit value" because the caller uses a name associated with the argument to imply a value which does not appear explicitly. A complete least of terms which describe a feature provided by more than one language follows:

Positional Value *(common name: N/A)*: These values are associated with arguments based on the index of the value in the list of all positional values.

Default Value *(common name: Optional Argument)*: A value that an author associates with an

[2] I intend to incorporate this information into the references in the final version of this paper, but I have not used Latex before so I will need to learn how to do that.

argument for use when the caller declines to supply a value.

Named Value *(common name: Keyword Argument, Option Argument* [3] *)*: These values are associated with an argument based on a name which the caller attaches to the value. The argument may be associated with several **synonymous** names.

Implicit Value *(common names: Switch, Flag, Option)*: An argument has an implicit value if the caller can specify a name associated with the argument but omit the value. For example, the ubiquitous "–help" flag in most CLI tools is an implicit value. An argument with an implicit value may also have **antonyms**, which invert the semantics of the primary name.

Typed Value *(common name: N/A)*: Some languages allow or require associating a type with an argument, in which case the value provided by the caller must be compatible with that type.

Destructuring *(common names: Pattern, Unpacking)*: Some languages provide a mechanism to break a composite structure (which might be a struct or a container such as a list) into its component parts. In some cases this functionality is available to an author, who can declare that an argument should have a specific type but the values within that type should be bound to separate local variables. In other cases it might be available to a caller, in which case the values within a composite structure are associated with different arguments. It is possible for a language to provide both of these variants simultaneously.

Variadic Function *(common name: N/A)*: A function which can accept an arbitrary number of values. Variadic functions are distinct from arguments with default values because with default values the author controls the range of acceptable argument counts. With variadic functions, the caller can pass in an arbitrarily large number of arguments (physical limitations notwithstanding). Variadic values are values which are associated with this feature. For example, a variadic function may require one positional value at the beginning of the value list. The first value is not a variadic value but all other values are. A variadic variable is provided by some languages. This is a local variable bound to some (possibly empty) aggregate datatype (typically a list) which contains variadic values.

[3] The term "option argument" is not to be confused with the term "option**al** argument". The former is used by shell users to refer to an argument that come after an option (such as "--ignore" "foo" in the previous example) while an option**al** argument is a term used by scripters and system programmers to refer to an argument associated with a default value.

Part I
Assessment

2 Overview

Most of this part is dedicated to describing the features of various languages that are relavent to this paper, in terms defined by this paper. Each of these sections will start with a summary of the features provided by the language. Next, it will explain the background knowledge necessary in order to understand the feature descriptions. This explanation will include an example of a "simple function and call" which serves two purposes. First, it provides the reader with a concrete example of what analysis looks like in the context of this language. For example, in Python "analysis" means assessing the behavior of the interpreter and the bytocode emitted by the compiler. Second, it provides a point of comparison when describing features. In the Python analysis, the description of destructuring discusses how the bytecode for this feature differs from a call that does not include destructuring. This keeps each section shorter because they do not have to explain the baseline they are being compared to. It also makes it easier for a reader to focus only on the features that interest them because they can read the simple call section then the feature section without having to pick through other features to find the background information that they require. Finally, each language analysis will describe each of the features.

Each feature description will start either with a statement that the feature is not provided or a statement of how the feature is provided. For example, in Python positional parameters are the "default behavior". Tutorials (including the official tutorial contained in the repository) commonly introduce functions by using positional parameters and introduce "keyword arguments" as a separate feature at a later point. Therefore, the Python section about positional values starts with "provided by default" while the section about named values starts with "provided through keyword arguments". This helps the reader understand the feature's relationship to the language and clarifies what terminology they should expect to see if they are reading language documentation, or what terms they should use if they want to perform a digital search for more information. After this statement, any distinctive qualities of the feature as provided by the language will be noted.

Finally, there will be 3 subsections: Syntax, Implementation, and Historical Record. All 3 of these sections inform the implementation of the mechanism and perform a service for the reader [4].

The syntax section explains what the source code looks like when the feature is used. This helps readers who are unfamiliar with the language in question understand the code snippets in the following subsections.

The implementation section explains the language behavior which causes the feature to be provided. This helps clarify the "concrete things" that the argument is associating.

The historical record section discusses, where possible, the motivation for the feature and lessons learned from implementation and community response. This gives the reader context about the environment the feature exists in, deepening their understanding.

The final section synthesizes the information from the descriptions. When different communities have similar concerns it will merge these concerns into a single description. When the concerns are distinct it will clarify the distinction. It will also look for opportunities to apply solutions created by one community to a concern raised by another. Finally, it will provide a compact table listing all concerns. The mechanism will either address each of these concerns or provide justifications for leaving specific concerns unaddressed.

3 Python

Python provides the following features:

- Positional Value

[4] It should go without saying that the author is also a reader, albeit a particularly invested one. =)

  - Callers typically decide between naming or positioning values but authors can restrict this decision.
  - Positional values must precede named values.

- Default Value
- Named Value
- Typed Value
- Caller Destructurig
  - Restricted to iterables and dictionaries.
- Variadic Functions
  - One or two variadic variables will be bound to a list for positional values and/or a dictionary for named values.

Python used to provide author destructuring for tuples, but this was removed in version 3.0.

3.1 Bytecode

Python has a compiler which produces bytecode [3, internals/compiler.rs section "Abstract"], and an interpreter which executes the bytecode [3, internals/interpreter.rst section "Introduction"]. Argument and return values are given using a stack managed by the interpreter; instructions may modify or move these values, even if this is not the primary purpose of the instruction [3, internals/interpreter.rst section "The Evaluation Stack"].

There are 2 families of instructions that are used throughout the examples in this section. The `LOAD` family puts values on the stack from different locations depending on the instruction. The `CALL` family initiates a function call after the stack has been prepared. There are also example-specific instructions which will be explained alongside the relevant example.

Note: This section omits bookkeeping instructions that are not topically relevant. For example, when calling a non-method function (one which is not associated with an object instance), the interpreter pushes `NULL` onto the stack before pushing the function. This instruction, and similarly uninteresting instructions, are omitted for brevity.

3.1.1 `LOAD` family

Instructions prefixed with `LOAD_` retrieve a value from some location (depending on the instruction) and put it onto the stack. Each instruction recieves an integer which represents an index into a C-level array. Which array is referenced depends on the instruction. When the array contains variable names, the instruction also retrieves the value associated with that name. The below table explains the contents of the array that each instruction references.

`LOAD_CONST`	Constant values which appear literally or implicitly in source code [2, Doc/library/dis.rst lines 964-966].
`LOAD_FAST`	Names of local variables which are guaranteed to be initialized. [2, Doc/library/dis.rst lines 1253-1259, Lib/inspect.py line 514]
`LOAD_NAME`	Names of non-local variables. If a local variable exists with the same name as a non-local variable then the value bound to the local variable will be returned. [2, Doc/library/dis.rst lines 969-972, Lib/inspect.py line 511].

3.1.2 `CALL` Family

These instructions tell the interpreter to call a function. This paper views this family as rooted in the plain `CALL` instruction, with all others being variants on this core instruction. When it needs to reference a behavior which occurs when a function is called, it examines only the plain `CALL` instruction and assumes that other instructions behave similarly unless the purpose of the variant is to change that specific behavior.

`CALL` Receives an integer indicating the number of argument values provided by the caller. The stack will contain the function to call followed by the argument values in separate stack locations. [2, Doc/library/dis.rst lines 1398-1410 Python/ceval.c lines 1314-1536].

3.2 Simple Function and Call

```
def add_values(a, b):
    return a + b

add_values(1000, 1001)
```

The bytecode generated for the call to `add_values` performs 3 tasks. First, it pushes the function onto the stack. Next, it pushes literal values which will become associated with arguments. Finally, it calls the function.

```
LOAD_NAME  0 (add_values)
LOAD_CONST 1 (1000)
LOAD_CONST 2 (1001)
CALL       2
```

The bytecode generated for the definition is similar. It uses `LOAD_FAST` (instead of `LOAD_NAME`) to refer to the variables associated with arguments and `BINARY_OP` (instead of `CALL`) to use the built-in `+` operator.

```
LOAD_FAST    0 (a)
LOAD_FAST    1 (b)
BINARY_OP    0 (+)
```

3.3 Positional Value

Provided by default. Generally, callers can choose whether to provide values by name or position when they make the call. All positional values must precede all named values [5, reference/expressions.html section 6.3.4 "Calls"]. Function authors can specify that some arguments with only receive their value by position. These are referred to as "positional-only arguments".

3.3.1 Syntax

Callers specify positional values by providing a comma-separated list of values. Function authors specify positional-only arguments by listing a literal `/` after the final positional-only argument [5, reference/compound_stmts.html section 8.7 "Function Definitions"]:

```
def add_values_mixed(position, /, either):
    return position + either

# valid
add_values_mixed(1000, 1001)
add_values_mixed(1000, b=1001)

# invalid: the value for argument
# "position" cannot be given by name
# add_values_mixed(position=1000, \
#                  either=1001)
```

3.3.2 Implementation

The bytecode for positional values is identical to the bytecode for the simple call. Python stores the values of local variables in the C-level array 'localsplus'. The `CALL` instruction copies positional arguments from the stack into this array [2, Python/ceval.c lines 1341-1353].

The bytecode for function definitions is identical regardless of the presence of positional-only arguments. The restriction is enforced within the `CALL` instruction. In particular, the helper function `positional_only_passed_as_keyword` uses the `co_posonlyargcount` and `co_localsplusnames` members of the code object. These variables track the number of positional-only arguments [2, Doc/library/inspect.rst lines 180-181] and the names of all arguments [2, Include/cpython/code.h line 155] respectively. If any names of positional-only arguments appear as keyword arguments then the helper raises an error. [2, Python/ceval.c lines 1182-1244]. Note that the helper is only called if the function does not include a variadic variable for named values [2, Python/ceval.c lines 1417-1431].

3.3.3 Historical Record

Positional values have always been available in Python and requires no special syntax to use.[5]

PEP 570 introduced positional-only arguments [4, peps/pep-0570.rst]. It gives several justifications for the change, most of which are concerned with maintaining a healthy ecosystem. There are two relevant [6] ecosystem harms the PEP is concerned with: inappropriate use of value names by callers and increased maintenance burden for library authors.

Inappropriate use of value names includes using non-meaningful names, such as a math function that takes one argument (the `sqrt` function takes one argument named `x`). It also includes providing values in an illogical order, such as calling the `range` function and supplying the `stop` value before the `start` value.

The increased maintenance burden occurs because all argument names are automatically and irrevocably added to the API surface of all libraries. It could be the case that a library author wants to implement a change which should be non-breaking in principle, but prompts a variable name change for clarity. This variable name change transforms the overall change into a breaking change.

The PEP is also concerned with functions that include a variadic variable for named parameters. For these functions, any non-variadic variables restrict the domain of the variadic variable, as their names will be associated with the distinct variable rather than the variadic one.

Finally, the PEP notes the curious case of the `range` function, which the PEP describes as accepting "an optional parameter to the left of its required parameter." In particular, if the `range` function only receives a single argument it is interpreted as the end of the range, but if it receives 2 arguments then the first is interpreted as the start while the second is interpreted as the end. This concern does not appear to be addressed by PEP 570 [7].

3.4 Default Value

Provided by default argument values.

3.4.1 Syntax

Function authors can define a default value by adding a literal `=` after the name of an argument, then the value [5, reference/compound_stmts.html section 8.7 "Function Definitions"].

```
def add_values(mandatory, optional=2000):
    return mandatory + optional

# valid
add_values(1000)
add_values(1000, 1001)

# invalid: the first argument is not
# associated with a default value
# add_values()
```

3.4.2 Implementation

Default values do not impact the bytecode generated for the definition or the call: they are both identical to the simple call. Instead, the `CALL` instruction retrieves default values from the code object and uses them when necessary [2, Python/ceval.c lines 1314-1536, trace_call_TO_initialize_locals].

3.4.3 Historical Record

Default values were added in version 1.0.2 [2, Misc/HISTORY lines 32809-32811]. There is additionally a note that default values "would now be quite sensible" in the version 0.9.4 release notes. This

[5] Unfortunately, positional values are assumed to be the default method of passing arguments by most programmers, including language authors, so there is no good citation for this assertion.

[6] There are also several concerns mentioned which are specific to Python and/or its implementation. For example, it references PEP 399 which requires that pure Python code and C extensions have the same expressive power. While important to the Python community, these concerns are not relevant to this paper.

[7] At least, I do not see anything that addresses it when I read the PEP and the implementation of range still inspects the number of provided arguments manually [2, Objects/rangeobject.c lines 81-120].

version changed argument processing so that functions receive all arguments as separate values, rather than as a single tuple [2, Misc/HISTORY lines 34550-34639].

PEP 671 proposes adding a feature which would allow function authors to provide an expression which will produce a default value at call time ("late evaluation") [4, peps/pep-0671.rst]. Currently, default values must be static/constant values which are determined when the function is defined. The mailing list discussion includes several disagreements, including whether or not it is appropriate for a function signature to contain un-inspectable objects and technical difficulties about scoping rules for late evaluated values. The proposal is still in the "draft" state, so it might be added in the future (possibly after trivial or significant changes to the proposal), but there has been no activity on the mailing list since 2021. [7]

3.5 Named Value

Provided through keyword arguments. Generally, callers can choose whether to provide values by name or position when they make the call. All named values must proceed all positional values [5, reference/expressions.html section 6.3.4 "Calls"]. Function authors can specify that some arguments will only receive their value by name. These are referred to as "keyword-only arguments".

3.5.1 Syntax

Callers provide named values by writing first a symbolic name, then a literal `=`, then the value. [5, reference/expressions.html section 6.3.4 "Calls"]. Function authors specify keyword-only arguments by listing a literal `*` before the first keyword-only argument. [5, reference/compound_stmts.html section 8.7 "Function Definitions"].

```
def add_values_mixed(either, *, named):
    return either + named

# valid
add_values_mixed(named=1001, either=1000)
add_values_mixed(1000, named=1001)

# invalid: the value for argument "named"
# must be given by name
# some_function(1000, 1001)

# invalid: named values cannot appear
# before positional values
# some_function(named=1001, 1000)
```

3.5.2 Implementation

The bytecode for function definitions is identical regardless of the presence of keyword-only arguments. The restriction is enforced within the `CALL` instruction. In particular, the helper function `initialize_locals` checks that the number of positional arguments is not more than expected [5, Python/ceval.c lines 1458-1462, trace_call_TO_initialize_locals], by checking the `co_argcount` member which tracks the number of arguments which may be positional [2, Doc/library/inspect.rst lines 146-149]. [2, Python/assemble.c line 556].

Bytecode for calls which use named values differ significantly from the simple call. For example, consider this code:

```
def add_values(a, b):
    return a + b

add_values(b=1001, a=1000)
add_values(1000, b=1001)
```

The bytecode for the first call differs from the simple call by adding the `KW_NAMES` instruction prior to the `CALL` instruction:

```
LOAD_NAME                1 (add_values)
LOAD_CONST               6 (1001)
LOAD_CONST               5 (1000)
KW_NAMES                 8 (('b', 'a'))
CALL                     2
```

`KW_NAMES` marks the given constant, in this case the tuple `('b', 'a')`, as a set of argument names to be

used by CALL [2, Python/bytecodes.c lines 2601-2605, 2644, 2869–2692, 2706–2709]. Then, the CALL instruction determines which values belong to which arguments by corresponding their respective positions on the stack and in the tuple. [2, Python/ceval.c lines 1383-1384].

The process is similar when some values are provided by position and others by name. The second call above does not have any additional instructions to handle this case:

```
LOAD_NAME                2 (add_values_mixed)
LOAD_CONST               5 (1000)
LOAD_CONST               6 (1001)
KW_NAMES                12 (('named',))
CALL                     2
```

The CALL instruction infers which value is named based on the restriction that positional values must precede named values [2, Python/ceval.c lines 1383-1384].

3.5.3 Historical Record

Named values were first introduced in Python 1.3 [1, Doc/tut.tex lines 3540-3626]. The feature was based on the similar feature provided by Modula-3 [1, Doc/tut.tex lines 3584-3586]. While keyword-only arguments (discussed later in this section) were added afterwards, the core syntax and semantics of keyword-only arguments have remained unchanged.

PEP 3102 introduced keyword-only arguments. It provides a single justification for the change: variadic functions cannot make use of default values. The PEP gives the following example:

```
def sortwords(*words, case_sense=False):
    pass
```

If the value associated with case_sense can be provided positionally then it must be provided in every call even if the caller wants the default value of False [8].

[8] Unless the caller wants to sort the empty list. =)

3.6 Implicit Value

This feature is not provided by Python.

3.7 Typed Value

Provided through type hinting. The Python compiler and interpreter do not change their behavior based on type hints. However, they do guarantee that the hints will be available to external tools and provide supporting infrastructure to help the tools work correctly. Both static analyzers and runtime checkers can make use of annotations.

3.7.1 Syntax

Type hints are specified using function annotations, as defined in PEP 3107. This means that function authors add a colon and type name after the variable name associated with the argument:

```
def typed_argument(x: str):
  pass
```

3.7.2 Implementation

Annotations are stored as metadata in the Python object which represents the function. Libraries can access them through the __annotations__ property, which contains a dictionary [4, peps/pep-3107.rst, "Accessing Function Annotations"].

3.7.3 Historical Record

The foundations for type hinting were added in PEP 3102, which defines the syntax for function annotations [4, peps/pep-3102.rst]. The Python developers then waited for external community-driven tools to experiment with different type-checking approaches. Eventually, they took lessons learned from the community and created a set of recommendations in PEPs 482, 483, and 484 [4, peps/pep-0484.rst, "Abstract"]. Much of their content addresses type theory issues, such as generics, variance, and special types like Any. Since this initial introduction there have been a number of PEPs which further clarify best practices or provide syntactic improvements to type

specifications. However, the core mechanism that this paper is concerned with - associating a type with an argument, regardless of how that type is specified - remains unchanged.

3.8 Destructing (caller)

Provided through argument unpacking. Caller destructuring is only available for iterables and mappings.

3.8.1 Syntax

This feature allows a caller to prefix one or more iterables with * in order to translate their contents into a set of positional values, and/or prefix one or more mappings with ** to translate their contents into a set of named values. For mappings, keys must strings naming an argument. [5, reference/expressions.html section 6.3.4 "Calls"]

```
def add_values(a, b, c):
    return a + b + c

# all of the below calls are equivalent
# to this:
# add_values(1000, 1001, 1002)

# destructure an iterable into positional
# values
l = [ 1000, 1001, 1002 ]
add_values(*l)

# destructure multiple iterables into
# positional values
first_part = [ 1000 ]
second_part = [ 1001, 1002 ]
add_values(*first_part, *second_part)

# destructure a mapping into named
# values
d = { 'a': 1000, 'b': 1001, 'c': 1002 }
add_values(**d)

# destructure multiple mappings into
# named values
first_part = { 'b': 1001 }
second_part = { 'a': 1000, 'c': 1002 }
add_values(**first_part, **second_part)
```

3.8.2 Implementation

The difference between the simple call and a call which includes destructuring is best explained by starting with the final instruction. While the simple call uses the plain `CALL` instruction a destructuring call uses the `CALL_FUNCTION_EX` instruction. `CALL_FUNCTION_EX` receives either 0 or 1 which tells it whether or not there is a mapping to destructure [2, Doc/library/dis.rst lines 1398-1410].

When it receives 1, there is a mapping to destructure which will be on the top of the stack. While the caller can use any mapping (and any number of mappings), `CALL_FUNCTION_EX` will always see a single dictionary when it executes (the process which ensures this is discussed in more detail later in this section). The dictionary is turned into a set of keyword arguments by interpreting the keys as names identifying arguments. [2, Objects/call.c lines 1029-1053]

The next item on the stack is an iterable to destructure. In this case, `CALL_FUNCTION_EX` might see any iterable on the stack. If the iterable is not a tuple it will convert it into a tuple [2, Python/bytecodes.c lines 3198-3207]. The elements of this tuple will be used as positional values. [2, Python/bytecodes.c line 3219].

When `CALL_FUNCTION_EX` receives 0 the process is similar, except that the top element of the stack is an iterable and there is no mapping.

The compiler ensures that `CALL_FUNCTION_EX` only receives dictionaries (rather than the arbitrary mapping object provided by the caller) with two instructions. First, it issues a `BUILD_MAP` instruction to place a new dictionary on the stack [2, Doc/library/dis.rst lines 1015-1023]. Then it adds the keys and values of each mapping object into this dictionary by repeatedly calling the `DICT_MERGE` instruction. For example, this code:

```
add_values(*d)
```

Compiles to this bytecode (note that BUILD_MAP receives the value 0 to indicate that it is building an empty dictionary):

```
LOAD_NAME          0 (add_values)
LOAD_CONST        13 (())
BUILD_MAP          0
LOAD_NAME          3 (d)
DICT_MERGE         1
CALL_FUNCTON_EX    1
```

When named values are provided separately from the destructured values, the freshly created dictionary is prepopulated with those values. For example, this code:

```
d = { 'b': 1001, 'c': 1002 }
add_values(a=1000, **d)
```

Compiles to this bytecode (note that in this case, BUILD_MAP receives the value 1 to indicate that there is one key-value pair on the stack):

```
LOAD_NAME          1 (add_values)
LOAD_CONST        17 (())
LOAD_CONST        11 ('a')
LOAD_CONST         3 (1000)
BUILD_MAP          1
LOAD_NAME          4 (d)
DICT_MERGE         1
CALL_FUNCTION_EX   1
```

When the caller provides multiple destructured iterables, or provides literal positional values in addition to one or more destructured iterables, the compiler issues instructions to merge them into a list, then converts that list into a tuple. For example, this code:

```
t0 = ( 1001, )
t1 = ( 1002, )
add_values(1000, *t0, *t1)
```

Compiles to this bytecode:

```
LOAD_NAME          1 (add_values)
LOAD_CONST         3 (1000)
BUILD_LIST         1
LOAD_NAME          4 (t0)
LIST_EXTEND        1
LOAD_NAME          5 (t1)
LIST_EXTEND        1
CALL_INTRINSIC_1   6 (INTRINSIC_LIST_TO_TUPLE)
CALL_FUNCTION_EX   0
```

If the caller provides only a single iterable to destructure, and no literal positional values, this iterable is placed onto the stack without modification and the tuple creation logic contained within CALL_FUNCTION_EX itself is triggered.

3.8.3 Historical Record

When argument unpacking was first introduced in version 1.6 [2, Misc/HISTORY lines 26740-26743], it only allowed callers to unpack a single iterable and/or a single mapping. For example, the call `add_values(*first_part, *second_part)` would have been illegal. PEP 448 expanded argument unpacking so that multiple values can be destructured in the same call [4, peps/pep-0448.rst]. The rationale given for this change was enhanced readability, as previously callers would either need to build iterables/dictionaries separately or destructure them manually, adding additional lines of code which are semantically sparse.

3.9 Destructuring (author)

While python does not currently support authorial destructuring, it did so prior to version 3.0 [4, peps/pep-3113.rst]. It allowed authors to declare that arguments should receive tuples whose values would be bound to separate local variables:

```
def distance((x1, y1), (x2, y2)):
  pass
```

This function would require that callers pass in 2 values which are both tuples containing 2 elements. The values from the first tuple would be bound to local variables `x1` and `y1`, while the values from the second would be bound to `x2` and `y2`.

The functionality was removed through PEP 3113. The rationale includes multiple implementation issues which are important to the Python community but not relevant to this paper.

3.10 Variadic Function

Provided by arbitrary argument lists and dictionaries. Positional values are collected by the former while named values are collected by the latter.

3.10.1 Syntax

Function authors specify variadic-ness by specifying the name for one or two variadic variables. The name for the variadic list must be prefixied by a * while the name for the variadic dictionary must be preceeded by a ** [5, reference/compound_stmts.html section 8.7 "Function Definitions"].

```
from itertools import chain

def add_values(*pos_vals, **named_vals):
    return sum(chain(pos_vals, \
                     named_vals.values()))

# All of these values appear in the
# pos_values list
add_values(1000, 1001, 1002, 1003)

# All of these values appear in the
# named_values dictionary
add_values(named_val0=1000,
           named_val1=1001,
           named_val2=1002,
           named_val3=1003)

# The values 1002 and 1003 appear in the
# pos_vals list while the names and
# values named_arg0=1000 and
# named_arg1=1001 appear in the
# named_vals dictionary
add_all_values(1002,
               1003,
               named_val0=1000,
               named_val1=1001)
```

3.10.2 Implementation

The interpreter tracks which positional values are also variadic values by checking the `co_argcount` variable associated with the function's code object. Remaining positional arguments are moved into the appropriate variadic variable, if it exists [2, Python/ceval.c lines 1355-1376]. It distinguishes variadic named values from non-variadic named values by checking if the name is expected; the interpreter already has to keep track of this information because an unrecognized value name is considered an error for non-variadic functions [2, Python/ceval.c lines 1378-1455].

3.10.3 Historical Record

PEP 468 updated the variadic variable for named values such that the author can retrieve the syntactic order in which the values were given. The rationale given for this change is that some users are developing APIs where order matters, such as serialization. [4, peps/pep-0468.rst]

Part II
Action

References

[1] Python Software Foundation. *Python 1.3 Source Code*. `https://legacy.python.org/download/releases/src/python-1.3.tar.gz`.

[2] Python Software Foundation. *Python 3.12.2 Source Code*. `https://github.com/python/cpython`.

[3] Python Software Foundation. *Python Developer Guide*. `https://github.com/python/devguide`.

[4] Python Software Foundation. *Python Enhancement Proposals (PEPs)*. `https://github.com/python/peps`.

[5] Python Software Foundation. *Python HTML Documentation*. `https://docs.python.org/3/archives/python-3.12.2-docs-html.zip`.

[6] Clark T et al. Katz DS Chue Hong NP. *Recognizing the value of software: a software citation guide*. `https://f1000research.com/articles/9-1257/v2`.

[7] *PEP 671 (late-bound arg defaults), next round of discussion!* `https://mail.python.org/archives/list/python-ideas@python.org/thread/UVOQEK7IRFSCBOH734T5GFJOEJXFCR6A/`.