Coral Programming Language Reference Manual

Rebecca Cawkwell, Sanford Miller, Jacob Austin, Matthew Bowers
rgc2137@barnard.edu, {ja3067, skm2159, mlb2251}@columbia.edu

October 15, 2018

Contents

1 Overview of Coral

The Coral programming language is an imperative and functional scripting language inspired by Python, with optional static typing used to enforce type safety and optimize code. It roughly aims to be to Python as TypeScript is to JavaScript. The basic syntax is identical to Python, and any valid Coral program can also be run by a Python interpreter. Coral also uses the newly introduced optional static typing syntax found in Python 3.7, allowing variables and function declarations to be tagged with specific types. Unlike in Python, these types are not merely cosmetic. The Coral typing system will be checked and enforced at compile time and runtime, preventing errors due to invalid type usage in large codebases, and will also be used to optimize the performance of the language where possible. The goal is to create a language that is as convenient as Python and as safe as an explicitly typed language with superior performance to existing scripting languages.

Our goals are:

- Python-style syntax with all its usual convenience, including runtime typing where types are not explicitly specified.
- Type safety where desired, with type specifiers on variables and functions enforcing correct usage in large code bases.
- Potential optimizations due to types known at compile time. If the argument and return types of a function are given explicitly, it can be compiled into a function potentially as performant as equivalent C code.
- Seamless interplay between typed and untyped code. You dont pay a penalty if you dont type your code, and typed functions can be called with untyped arguments and vice versa.

1.1 The Coral Type System

When writing scientific or production code in Python, certain operations involving nested loops or recursive function calls often become too expensive to run using pure Python. While Python supports extensions written in a low-level language like C, these are hard to maintain and interface poorly with Pythons object model. With Coral, these can be optimized by providing enough static type hints for the compiler to infer all types in a given function at compile time and produce an efficient machine code representation that can be run as fast as a lower-level language.

These hints will also prevent variable from being incorrectly passed to functions intended for usage with a different type, preventing errors that may be tricky to debug due to the flexibility of Python syntax. Functions intended to be called on strings will not accidentally be called on list with unintended consequences.

This typing system is often referred to as gradual typing in the compilers literature, and uses a combination of dynamic typing and type inference to determine types at compile time where possible, falling back to a standard dynamic typing system where types are too difficult to infer.

We hope this language will have potential practical usage in the real-world, and, with future development, we hope it could even become widely used as a Python compiler for production code.

2 Lexical Conventions

2.1 Comments

Coral has both single-line and multi-line comments. Any tokens following a # symbol are considered part of a single-line comment and are not lexed or parsed.

```
x = 25 # x is in inches
# x is the average length of a coral snake
```

Multiline comments begin and end with triple quotes.

```
"""
Coral snakes are known for their red, white and black banding
"""
```

Within triple quotes, single or double quotes can be arbitrarily nested.

2.2 Identifiers

Valid identifiers are made from ASCII letters and decimal digits. An identifier must begin with a letter, can contain an underscore and cannot be a Coral keyword.

```
# valid identifiers
pinkPython
GardenSnakeCount
snake_length
babysnake4

# invalid identifiers
25coral
5
```

2.3 Operators

Coral uses the following reserved operators:

```
+  -  <  >  <=  >=  !=  ==  *  /  =  **
```

2.4 Keywords

Keywords are reserved identifiers. They cannot be used as ordinary identifiers for other purposes. Corals keywords are:

```
if else for while def return and or in is not elif assert
pass continue break class print int str bool float
```

To ensure compatibility of Coral with Python, using unimplemented Python keywords returns a compile-time error. The unimplemented Python keywords are global, await, import, from, as, nonlocal, async, yield, raise, except, finally, is, lambda, try, with.

2.5 Indentation

Coral uses indentation (tabs) to determine the grouping of statements. A given lines indentation is determined by the number of tabs preceding the first character. Statements are grouped by indentation level. Control flow keywords like if, else, for, and while must be followed by a colon, and the subsequent lines included in that control flow must be indented at the same level, unless a further control flow keyword is used. This behavior is identical to Python. No line may be indended more than one level beyond the current indentation level.

```
for i in [1, 2, 3]:
        print(i)

if x == 3:
        x = 4

while x < 3:
        if x < 5:
                return x
        else:
                return x + 1
```

2.6 Separators

Coral uses parentheses to override the default precedence of expression evaluation, semicolons to separate two consecutive expressions on the same line, tabs to denote control flow, and newlines to separate statements.

```
x = (a + b) * c # overrides default operator precedence
x = 3; y = x + 1 # allows two expressions in one line
if x == 3:
    print(x) # control flow
```

2.7 Literals

Literals represents strings or one of Coral's primitive types: float, char, int, and boolean.

2.7.1 Float Literals

A float literal is a number with a whole number, optional decimal point, a fraction, and an exponent.

```
((([0-9]+\.[0-9]*)|([0-9]*\.[0-9]+))((e|E)(\+|-)?[0-9]+)?|[0-9]+((e|E)(\+|-)?[0-9]+))
```

Examples of float literals:

```
25
2.5
0.000407
24.
1e+3
12.6E-2
```

2.7.2 String Literals

A string literal is a sequence of characters enclosed in single or double quotation marks, i.e. abcdefghijklmnopqrstuvwxyz. The matching regex is

```
("[^"'\\]*(\\.[^"\\]*)*")|('[^"'\\]*(\\.[^'\\]*)*')
```

Example of string literals:

```
"Hello world"
'Here are 4 words'
```

2.7.3 Char Literals

If a string literal contains only one character and is assigned to a variable of type char, it becomes a char literal. Char literals cant exist anonymously as other literals can, and must be bound to a variable. This is done to minimize departure from Python syntax which does not contain chars.

```
x: char = 'a'          # char
x = 'a'                 # NOT a char. This is a string literal
```

2.7.4 Int Literals

An integer literal is any sequence of integers between 0 and 9. The matching regex is [0-9]+.

2.7.5 Boolean Literals

Boolean types represent true and false. They are represented in Coral by the True and False keywords.

3 Data Types

Coral represents all pieces of data as either an object or a primitive.

3.1 Primitives

Primitives are series of bytes of some fixed length, and there are four primitive types in Coral:

int (4 bytes, 2's complement)

float (8 bytes, IEEE standard double)

char (1 byte, ASCII)

bool (1 byte, 00000001 for true, 00000000 for false)

Note that there is no separate double type.

3.2 Objects

Any piece of data that can't be inferred to be one of the primitive types is represented as an object. An object has an associated type and an associated value. The value can contain primitives and/or references to other objects. How to interpret the content of the value of an object is determined by its type. References are completely under the hood and not user-accessible as pointers.

3.3 Mutability

The primitive objects in Coral are immutable, including ints, floats, and booleans. Strings are not primitives, but are also immutable. All operations which modify these objects actually return new objects. All user defined types, as well as those in the standard library, are mutable, in the sense that modifying them does not overwrite the underlying object. Assigning to any variable will still overwrite its underlying object, i.e. x = 3; x = 4 assigns an integer literal with value 3 to the variable x, and then assigns a different integer literal with the value 4 to the same variable. Likewise, x = [1, 2, 3]; x = [4, 5, 6] will assign one object of type list to x, and then assign a different object to it. On the other hand, x = [1, 2, 3], x[1] = 4 will modify the underlying data associated with the variable x, returning x = [1, 4, 3].

While strings are immutable, explicitly initialized arrays of chars are mutable. When explicitly declared as a char array, string literals can be used to initialize these arrays. For example,

```
1  char[] x = "hello"
```

is a valid expression with type char[], interpreted by the compiler as char[] x = [h, e, l, l, o].

3.4 User-Defined Types

Users can define their own types through the mechanism of classes. Inheritance is not implemented. Classes are declared using the typical Python syntax as

```
1  class Foo:
2          def __init__(self, arg1, arg2, . . .):
3                  self.arg1 = arg1
```

```
                . . .

        def method(self, x):
                return x + 1
```

More will be said about these classes later. They are generally mutable, and can be used like any other objects in Coral.

3.5 Standard Library Types

Coral provides a number of built-in types, including lists and strings. These will be expanded on in a later section, but are implemented in Coral as classes, and have associated methods and properties. Coral strings are immutable and support a rich variety of operations. They are fully memory safe. Likewise, lists are dynamic arrays which can contain objects of any type and can be modified with functional or imperative syntax.

```
x = "Hello" # this is a string
y = "World"
z = x + " " + y # z is now "Hello World"

x = [1, 2, "h"]
x[2] # returns "h"
```

3.6 None

None is keyword that returns a reference to a predetermined object of a special type. There is only ever one object of this type.

3.7 Memory Model

Coral provides a simple reference-counting garbage collection mechanism which frees memory allocated to objects which no longer have references attached to them. No explicit memory management is required.

In Coral, with a few exceptions due to typing optimizations, all names are simply identifiers used to look up data in a hashtable. All function calls are "pass by name", and do not involve copying data. Thus the user does not have to worry about passing an object "by reference" or returning "by reference" or "by value". Thus, for example

```
x = [1, 2, 3]
y = x
y[1] = 5 # x and y are now both [1, 5, 3]
```

or similarly

```
def foo(x):
    x[1] = 5

y = [1, 2, 3]
```

```
5
6  foo(y) # y is now [1, 5, 3]
```

On the other hand, assignment always acts as an assignment operator, and will not change the value of a bound variable, i.e.

```
1  def foo(x):
2      x = 5 # assigns a new value to x in the local scope
3
4  y = 3
5
6  foo(y) # y is still 3
```

This behavior is universally consistent with Python.

4 The Coral Type System

4.1 Overview

The Coral language uses a gradual typing system which combines type inference with dynamic typing. Even in the absence of static type hints, the Coral language makes an effort to use gradual type-inference algorithms inspired by Hinley-Milner to infer types, optimize code, and raise errors where invalid types have been used. In the presence of explicit types, further optimization and type checking can be performed, improving the safety and performance of code even further.

All explicit type hints are guaranteed, and errors will be raised for any invalid type usage either at compile or runtime. While Coral strives for a relatively error-free runtime experience, as a language without mandatory static typing, many errors cannot be caught at compile time. These issues will still be caught by the runtime environment, and the execution of the program will abort when these are detected. This behavior can be enabled or disabled at compile time. When types cannot be inferred, Coral will behave exactly like Python as a dynamically typed language with generic objects passed to all functions and hash-lookups performed at runtime to determine which function to execute.

4.2 Explicit Typing

Variable declarations, formal parameters, and return values can be marked with explicit types, allowing for compile-time and runtime type-checking and enabling the compiler to optimize performance. Typed variables are denoted with a colon followed by a type, like int, float, str, char, or a user defined class.

```
1  x : int = 3
2  x : str = "Hello World"
3  x : char = 'a'
4  x : char = "a" # also works
5  x: MyClass = MyClass(5)
```

Once an identifier has been associated with a class, it cannot be assigned to an object of a different class. Hence,

```
x = 5 # x is an int for the moment
x = "not anymore" # note that this is legal Python (and Coral)

x : int = 5 # x is an int forever and ever
x = "this isn't allowed in Coral"
x = 4 # but this is allowed
```

4.3 Optimization

When the compiler can infer that a piece of data will consistently be of a certain primitive type, it unboxes the data, converting it to a primitive instead of a generic object with type determined at runtime.

As much as possible, typed code does not behave fundamentally differently from untyped code, even if that leads to a performance penalty relative to pure C. For example (list) index bounds will still be checked at runtime, unlike in C. This means the runtime performance wont be quite as optimized as C, but removing these runtime checks when typing code would be confusing for the user.

4.4 Typed Lists

Lists are a built-in class that stores a series of pieces of data in an array. When every element of a list can be inferred to be of the same type and that type is a primitive type, usually the array is converted from an array of references, which can point to objects of various types, to an array of primitives, which must all be of the same type. Lists don't behave exactly like C arrays; Python-style conveniences like bounds-checking remain to ensure Python memory safety remains intact.

It is difficult for the compiler to determine that every element in a list is the same type, even if most or all non-list variables in a program are explicitly typed, so if object-to-primitive conversion in a list is desired, best practice is to type it explicitly.

```
my_list = [4, 3, "1"] # my_list contains an array of references (to objects of different types)

my_list = [4, 3, 1] # still array of references, which all just happen to be of the same type

int[] my_list = [4, 3, 1] # array of ints
```

Lists of non-primitive types always contain arrays of references, not arrays of the values of the pointed-to objects.

```
MyClass[] my_list = [...] # my_list is always an array of references
```

Note: when you access an item in a list, de-referencing happens under the hood and doesn't require any action on the part of the programmer.

4.5 Function Typing

A function can be declared with or without explicit typing. Functions with explicit typing have their typing behavior absolutely guaranteed, either at compile or runtime. Coral makes every effort to check the types of function calls at compile time, but if that is not possible, they will fail when

the code is executed at runtime. A function in which every type can be inferred can be compiled to efficient machine code. Thus this explicit typing is highly recommended for bottleneck code.

For example, this function takes two integer arguments and returns a string:

```
def foo(x : int, y : int) -> str:
    if x == 3:
        return "hello"
    elif y == 3:
        return "goodbye"
```

This function can be compiled to efficient code as a result of the copious type hints. It will fail if called with invalid arguments, i.e.

```
foo(3, "hello") # fails
x : int = foo(3, 4) # fails
```

4.6 Typing with User Defined Types

User-defined types can still be type-inferred. This is done using a similar method to build-in classes like strings. The model closely resembles TypeScript's type inference model, which looks at fields and methods which have been called on an identifier and matches them to possible user-defined types.

5 Statements and Expressions

5.1 Statements

A Coral program is made up of a list of statements (stmt). A statement is one of the following:

```
Expression
Class declaration
Function definition
Return statement
If statement
If-else statement
For loop
While loop
```

Blocks of statements can be enclosed using indentation as discussed earlier.

5.1.1 If-Else Statements

If statements consist of a condition (an expression) and a series of statements. The series of statements is evaluated if the condition evaluates to True. If the condition evaluates to False, either the program continues or an optional else clause is executed. In pseudocode, this is:

```
if condition:
        # series of statements
```

```
else:
    # other series of statements
```

5.1.2 While Statements

Similar to a for statement, a while statement consists of a condition and a series of statements. The statements are repeatedly evaluated as long as the condition remains True before each repetition.

```
while condition:
        # series of statements
```

5.1.3 For Statements

For statements in Coral iterate over lists. They consist of a looping variable, an instance of a list to be looped over, and a series of statements. The series of statements is evaluated for each item in the list in order, where the looping variable is assigned to the first element of the list for the first iteration, the second for the second iteration, etc. Note that assigning over the looping variable will not change the original variable in the list.

```
for x in some_list:
        print(x)
        x = 1 # this will not affect some_list
```

5.2 Expressions and Operators

Expressions are parts of statements that are evaluated into expressions and variables using operators. These can be arithmetic expressions which includes an operand, an arithmetic operator, an operand, and a function call, which calls a function and returns a value.

5.2.1 Unary Operators

Coral has two types of operator: uop and operator.

Uops are unary operations that act on an expression. In Coral, the unary operators are NEG and NOT, i.e. ”-” and ”not”.

NEG negates an integer or floating point literal, as in

```
x = -5 # represents the negative number 5
```

NOT represents negation of a boolean expression, but can also be applied to integers and floating point numbers, where any non-zero number is considered to be True, and False otherwise.

```
a = True
b = not a # b equals False
c = not a # c equals False
c = not 5 # c equals False
```

5.2.2 Binary Operators

The following list describes the binary operators in Coral. All of these operators act on two expressions:

```
| Binop of expr * operator * expr
```

Unless otherwise stated, they act only on primitives. We do not currently plan to support operator overloading for user-defined types, but that is subject to change.

1. Assignment Operator

   The assignment operator stores values into variables. This is done with the = symbol. The right side is stored in the variable on the left side. This can be applied to any type.

```
x = 3.6 # 3.6 is stored in the variable x
```

2. Arithmetic Operator

   (a) Addition is performed on two values of the same type. This can be applied to strings.

```
5 + 10 # Evaluates to 15
15.1 + 14.9 # Evaluates to 30
"hello" + "world" # Evaluates to "helloworld"
```

   The Coral language also permits automatic type conversion between integers and floating point numbers. For example

```
5 + 10.0 # Evaluates to 15.0
15.1 + 15 # Evaluates to 30.1
```

   (b) Subtraction is performed on two values of the same type, again with the possibility for automatic type conversion between integers and floats.

```
5-10 # Evaluates to -5
10.5 - 5.4 # Evaluates to 5.1
```

   (c) Multiplication is performed on two values of the same type, again with the possibility for automatic type conversion between integers and floats.

```
5 * 5 # Evaluates to 25
20.0 * .25 # Evaluates to 5.0
```

   (d) Division is performed on two values of the same type, again with the possibility for automatic type conversion between integers and floats.

```
10 / 2 # Evaluates to 5
10.8 / 2 # Evaluates to 5.4
```

   (e) Exponentiation is performed on two values of integer or floating point type.

```
        2 ** 3 # Evaluates to 8
        3.5 ** 2 # Evaluates to 12.25
```

3. Relational Operators

   Relational operators determine how the operands relate to another. There are two values as inputs and returns either true or false. These are the ==, !=, ¡, ¡=, ¿, ¿=.

```
    x = 1
    y = 2
    z = (x > y) # Evaluates to false
    z = (x < y) # Evaluates to true
    x == y # Evaluates to false
    x != y # Evaluates to true
```

5.3 Operator Precedence

The following is an operator precedence table for unary and binary operators, from lowest to highest precedence.

Operator	Meaning	Associativity
;	Sequencing	Left
=	Assignment	Right
.	Access	Left
or	Or	Left
and	And	Left
= !=	Equality / inequality	Left
>< >= <=	Comparison	Left
+ -	Addition / Subtraction	Left
* /	Multiplication / Division	Left
**	Exponentiation	Right
not	Negation / Not	Right

5.4 Functions

A function is a type of statement. It takes in a list of arguments and returns one value. The body of a function is delimited by indentation as described in the first section. Functions can be written either specifying or not specifying the return type. An example of these two types of function declarations are as follows:

```
def f_not_spec(x,y):
        returns x + y

def f_spec(x : int, y : int) -> int:
        if x == 1:
        return x
    else:
        return x + y
```

5.5 Function Calls

Functions are called using their identifier and arguments inside parentheses. For example:

```
f_not_spec(1,2)  # function call which evaluates to 3
f_spec(1,2)  # function call which evaluates to 1
```

If the function is called with invalid arguments, it will either fail to compile or fail at runtime with a TypeError.

5.5.1 Variable Assignment from Functions

A function may be called as the right-hand side of a variable assignment. The variable would be assigned to the return value of the function.

```
example = f_not_spec(1,2) # 3 would be assigned to example
```

6 Classes

Coral supports basic classes without any inheritance mechanism. Classes have instance variables, class variables, and class methods. Classes are declared using the typical Python syntax as

```
class Foo:
        def __init__(self, arg1, arg2, . . .):
                self.arg1 = arg1
                . . .

        def method(self, x):
                return x + 1
```

The __init__ method is a constructor which is called by default when a class is initialized. The self keyword must be explicitly passed to every method, and can be used to refer to the object itself. Classes are initialized with the following syntax:

```
x = Foo(3)
```

Class methods are called with the following syntax:

```
x.method(5) // returns 6
```

All user-defined classes are mutable, and arg1, arg2, and so on can be modified as desired. They behave like every other primitive object, but may have methods. Strings are implemented as a Coral class.

7 Standard Library

7.1 Lists

Coral has a built-in list data structure with dynamic length that can hold objects of arbitrary type. They behave identically to Python lists, and support the following operations:

Method/operator	Behavior
lt[n]	returns the nth element of the list
lt.append(x)	appends x to the list
lt.insert(i, x)	inserts x as the ith element in the list
lt.remove(x)	removes the first element of value(x) and returns true if x was present and false if it wasn't
lt.pop(i)	removes and returns the ith element in the list
lt.count(x)	returns the number of time x appears in the list

7.2 Strings

Strings are a class implemented in Coral, which supports indexing and a variety of useful methods for handling text-based data. Strings are immutable, so any operation on an existing string will return a new string.

Method/operator	Behavior
str[n]	returns the nth character in str
str1 + str2	returns the concatenations of str1 and str2
str.upper()	returns a string with all characters uppercase
str.lower()	Returns a string with all characters lowercase

7.3 range() and print()

range(i, n) returns a list of integers from i to n - 1.

```
r = range(1, 5)
print(r) # prints [1, 2, 3, 4]
```

If i is less than n, the list goes backward:

```
r = range(5, 1)
print(r) # prints [5, 4, 3, 2]
```

print(x) sends a string representation of x to standard output. The nature of that representation depends on the type of x. If x is one of the four primitive types (boxed or unboxed), there is a built-in print for it, e.g print(3) invokes print(x : int) -¿ str. If x is a non-primitive, x. __str__() is invoked if it is defined. Otherwise, the memory address of x is sent to standard output.

7.4 Casting

The user can explicitly cast objects from one type to another, for both typed and untyped variables. Explicit casts are must be used to convert between typed variables. Only certain classes have built-in casts from one to another. In particular:

```
int(x : float) -> int
float(x : int) -> float
str(x : int) -> str
str(x : float) -> str
str(x : char) -> str
char(x: str) -> char # string must be of length one
```

8 Sample Code

```
# GCD with static typing
def gcd (a : int, b : int) -> int:
    while(a != b):
        if(a > b):
            a -= b
        else:
                b -= a
    return a

# GCD without static typing, also valid
def gcd = (a, b):
    while(a != b):
        if(a > b):
            a -= b
        else:
                b -= a

# Printing a list with static typing
my_list : str[] = ["1", "1.0", "one"]
while(i : int < len(my_list)):
    print(my_list[i])
    i += 1

# Printing a list without static typing
my_list = ["1", "1.0", "one"]
while(i < len(my_list)):
    print(i)
    i += 1

# Declarations with static typing
x : int = 5
x : float = 3.4
x: char = "a"
bool x = true
x: str = "Hello World!"
x: int[] = [1, 2, 3]

# Declarations without static typing
x = 5
x = 3.4
x = "a"
```

```
x = true
x = "Hello World!"
x = [1, 2, 3]

# More examples
wont_compile : str[] = [1, 2]; # Elements in an array must be
        # the type of the array if it is specified

x : int = 1.0 # Error
x : int = int(1.0) # Okay
x = "this assignment produces a type error"

string1 = "Hello "
string2 = string1 + "World!"
print(string2) # Prints Hello World!

string3 = "Hello World!"
string3[0] # Returns "H"
string3[0] = "h" # this assignment produces an error,
        # since strings are immutable
```