**COMP 496**

**Pie 0.2 Language Specification**

**Jason Bell 3078931**

**February 15, 2016**

As Pie employs the .NET runtime, this document assumes familiarity with that environment.

**General**

Comments are written as in C-style languages:

// Single line comment

/\*

Multiline comment

\*/

Operators are as follows, in increasing order of precedence:

|  |  |  |
| --- | --- | --- |
| Operator | Associativity | Description |
| || | Left | Logical OR |
| && | Left | Logical AND |
| | | Left | Bitwise OR |
| ^ | Left | Bitwise XOR |
| & | Left | Bitwise AND |
| ==, != | Left | Equality, inequality |
| <, >, <=, >= | Left | Comparators |
| is, as | Left | Type comparison, cast to type |
| <<, >> | Left | Bitwise shifts |
| +, - | Left | Addition, subtraction |
| \*, /, % | Left | Multiplication, division, modulo |
| ++, --, ~, ! | Right | Increment, decrement, bitwise NOT, logical NOT |
| =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>= | Left | Operation, then assignment |

Variable declaration and type instantiation is as in Python, without C#’s var and new keywords:

X = SomeClass()

**Namespace Imports**

Namespace importing is as follows:

import System.IO

They may also be combined for brevity:

import System.IO,

System.NET

**Namespaces**

Namespaces can be declared individually or in combination:

namespace foo:

namespace bar:

namespace foo.bar:

**Classes**

In Pie 0.2, classes can be one of three forms: types, modules, and enums. Interfaces and structs are not included in the language. The ability to declare interfaces serves no purpose in a dynamic typed language. Existing interfaces can be inherited by Pie types if necessary. Structs, as value types, provide very important performance advantages over reference types. However, the late binding inherent to dynamic typing will completely cancel out any performance gain by using them. If a user must implement an interface or struct, they would be encouraged to do so in another language and import it into Pie.

Type and module methods default to public and fields default to private, unless a different modifier is specified. Types, modules, and enums can be declared as internal, public, or final.

Types are directly equivalent to C# non-static classes and are declared as:

type ClassName(Inherited types):

A Pie type may inherit other Pie types, C# classes, or interfaces. If an interface is inherited, the Pie type must explicitly implement its contract. Pie types declared on their own or that inherit another Pie type are dynamic: new members can be added during runtime. This is done through assignment to a member that is not already a member of the type. Pie types that inherit a class defined in a static typed language remain static typed for consistency: this is the same approach that IronPython takes.

Modules are equivalent to C# static classes, and are declared as:

module ModuleName:

All members of a module default to “shared”: Pie’s equivalent to the static keyword.

Enums behave the same as in C#: they may inherit a primitive type, and serve as containers for a series of constants:

enum EnumName(Inherited type):

a // 0

b // 1

c // 2

d // 3

Enum constants may have values assigned to them, and all following constants will increment from that one:

enum EnumName(Inherited type):

a // 0

b = 33 // 33

c // 34

d // 35

**Fields**

Fields are declared by an identifier in the class body:

type foo:

bar

The bar field of the foo type is declared as dynamic and so can have anything assigned to it, whether at declaration time or runtime. Fields default to private unless another modifier is specified.

type foo:

public bar = “something”

**Methods**

Methods come in two forms: acts and funcs. An act is a method that does not return a value, while a func does return a value.

type foo:

func Exp(x):

return x \* x

act Exp(x):

x \*= x

**If Conditionals**

If conditionals are as in Python, and may be multiple or single lined:

if condition DoSomething()

if condition:

DoSomething()

if condition DoSomething()

else DoSomethingElse()

if condition DoSomething()

else:

DoSomethingElse()

if condition:

DoSomething()

else:

DoSomethingElse()

**For Loops**

For loops are iterators much like those in C-style languages, adapted to a white-spaced context. Like C-style for loops, they have initialization, conditional, and step components. All loops may be exited or continued with the break and continue keywords.

for i = 0; i < 10; i++:

DoSomething(i)

**For Each Loops**

Pie for each loops behave the same as C# for each loops:

for item in itemList:

Item.DoSomething()

**While Loops**

Pie while loops behave the same as C# while loops:

while condition:

DoSomething()

**Switches**

Switches are defined as follows, and at present do not support fall-through case blocks:

switch n:

case 1:

DoSomething()

case 2:

DoSomethingElse()

else:

SomeDefaultAction()

**Tests**

Pie supports unit testing as an integrated component of the language. Inside the test, assert statements are used to test conditionals. If an assertion fails, the test fails:

test TestSomething

assert false //This will fail

An exception being thrown and not caught in a test will also cause that test to fail:

test TestDivision:

assert 1/0 // Divide by zero exception will cause test to fail

**Collections**

Pie does not support declaration of arrays, as any performance advantages gained by using them are nullified in a dynamic typed context. However, arrays created elsewhere can be used in Pie. Indexer access in Pie is the same as C#:

someList[i] = something

As a shortcut, Pie has keywords to create lists, queues, stacks, and dictionaries as containers for dynamic typed objects.

L = list() // Creates a list for dynamic typed objects.

Q = queue() // Creates a queue for dynamic typed objects.

S = stack() // Creates a stack for dynamic typed objects.

M = map() // Creates a dictionary for dynamic typed keys and objects.

**Generics**

For this iteration, Pie does not support generics as they are of limited use in a dynamic typed context: objects are already “generic”.