Pyre

A Pythonic, expression-oriented language implemented in Python.

Install

Standard install:

```
python3 setup.py install
```

If you are developing the module:

```
python3 setup.py develop
```

Usage

```
usage: ipyre [-h] [-a {repl,stdin,load}] [-f FILE]

optional arguments:
  -h, --help
    show this help message and exit
  -a {repl,stdin,load}, --action {repl,stdin,load}
    what to execute: the repl, from stdin or load a file.
  -f FILE, --file FILE
    the file to load
```

The current state of Pyre

Done:

- While loops
- Error handling (all errors)
- Function defitions and calls
- Attribute access
- Variable assignment and scoping
- Blocks, return and break

Todo:

- For loops
- Generators

- Error handling (specific errors)
- Immutable and mutable variable modifiers
- Most of the Pyre object space
- Most of the Pyre standard library
- Making fundamental types available in Pyre
- Python interop

Some examples

Factorial

```
let factorial = def (n)
    if n.equals(1)
        1
    else
        n.mul(factorial(n.sub(1)))
```

Validated user input

```
let get = def (prompt, validate)
    while True
        try
            return validate(input(prompt)) #early return breaks the loop
        except
            print("Invalid input, please retry.") #print error and iterate
```

Syntax overview

A Pyre program consists of a single *expression*, which is evaluated when the program is run. In order to evaluate more than one expression per program, you must use *blocks.* Because of this, whitespace does not matter, in fact, it is completely ignored by the parser. As such, this:

```
let
x
=

10
```

Is equivalent to this:

```
let x=10
```

An expression can be an assignment, a block, a conditional, a loop, a function definition, a try expression, a function call or an attribute access.

Blocks

Blocks allow you to evaluate more than one expression. The last expression in the block will be implicitly returned. A `return` statement will exit a block, returning a value.

```
do
    something
    something
    x #<- this is returned
end

do
    something
    return y #<- the block exits here and returns y
    x
end
```

Assignment

Assignments in Pyre are done via the `let` expression:

```
let var = value
```

Pyre is scoped lexically, and variables only propagate *down* scope, **not** up.

An assignment expression returns `value`.

Conditionals

Conditionals in Pyre use the `if` block. This takes two expressions: the condition and the expression to evaluate if it is true, as well as an optional else statement.

```
if cond expr
else elseexpr
```

This returns `expr` if `cond` is truthy, else it returns `elseexpr` if this does not exist, it returns `None`.

While loops

These take the form:

```
while cond body
```

They return a list of values accumuated from executing `body` every iteration.

Function calls

These take the traditional comma-seperated, bracketed syntax when there is one or more arguments. However, when there are no arguments, a exclamation mark is used instead, like so:

```
print!
```

Attribute access

Uses the traditional dot-operator syntax:

```
ident.attr
```

Try expressions

Use the syntax:

```
try expr
except eexpr
```

They return `expr` if no error occurred while evaluating it, and `eexpr` otherwise. Crucially, `expr` *will* be evaluated both times.

Function definitions

This uses the syntax:

```
def (args) body
```

Where `args` is a comma-seperated list of argument names and `body` is the expression to execute when the function is called, in terms of `args` and any variables up-scope.

It returns a function object, a first-class value.

Standard functions and methods

Object objects

Object#equals

Compares two objects.

Object#apply

Takes a function and returns the object applied to that function.

Object#str

Stringifies an object.

Object#setattr

Sets an attribute on an object.

Object#getattr

Gets an attribute of an object.

Number objects

Number#add

Adds two numbers, returning a third.

Number#sub

Subtracts one number from another, returning a third.

Number#mul

Multiplies one number by another, returning a third.

Number#div

Divides one number by another, returning a third.

Number#pow

Provides the exponentiation operation.

Number#int

Truncates a number, returning its integer portion.

Number#gt

Provides the 'greater than' relational operator.

Number#lt

Provides the 'less than' relational operator.

Number#and

Provides logical 'and'.

Number#or

Provides logical 'or'.

Number#not

Provides logical 'not'.

String objects

String#len

Returns a number containing the string's length.

String#num

Attempts to parse the string as a number, raising ValueError if it can't.

List objects

List#get

Gets the element at an index.

List#set

Sets the element at an index.

List#len

Gets the length of the list.

List#map

Applies a function to each element in the array, returning the accumulated values.

List#filter

Applies a function to each element in the array, returning only those, for which the function returns a truthy value.

True

A singleton value, exactly equal to 1.

False

A singleton value, exactly equal to 0.

quit!

Exits the program.

print(values*)

Prints a some values, seperated by spaces.

input(prompt?)

Displays a prompt and gathers the user's input.

list(values*)

Constructs a list from its arguments.