esc

Release 1.0.0

Nov 11, 2019

Contents

esc (pronounced /esk/) is an Extensible Stack-based Calculator designed for efficiency and customizability. What does this mean?

- esc is *stack-based*, operating using a Reverse Polish Notation (RPN)-like syntax. Rather than typing `2 + 2` and pressing an equals key, you enter the two numbers `2` and `2` onto *the stack*, then choose `+` to add them. This can be slightly awkward at first, but it means no parentheses are necessary, and for most people it becomes faster and more elegant than the standard algebraic method with a small amount of practice. In addition, it is considerably easier to customize and program.
- esc is *extensible*. If you frequently need to multiply two numbers together, add five, and then divide the result by pi, you can add a function to the calculator to do this specific operation using a couple of lines of Python code. The extension features are simple enough to be accessible even to people who do not know Python or have little to no programming experience.

  esc operations are arbitrary Python code, so if you want to get fancy, they can get arbitrarily complicated. You can even call APIs to perform calculations or get data!
- esc is *fast*, *simple*, and *terminal-based*. All you need is a working terminal (at least 80×24) and your keyboard.

CHAPTER 1

User Manual

The esc user manual describes how you install and use esc to perform calculations. If you're looking for information on hacking esc to add custom functionality, you want the *Developer Manual*.

1.1 Installation

The easiest way to install esc is through pip:

```
pip install esc-calc
```

This will install a package `esc` and an executable `esc` script, so you should be able to run esc like:

```
python -m esc
```

Or just:

```
esc
```

(This latter option may not work depending on your system configuration.)

You can also install esc from source. Find it on GitHub.

1.2 Terminology and Notation

Throughout the rest of this documentation, we will rely on some simple terminology and notation:

- In esc's interface, the stack window shows numbers from top to bottom. To avoid wasting large amounts of space in the documentation, we often describe the stack in terms of a Python list, with the leftmost element showing at the top, so a stack containing the numbers 1, 2, 3, and 4 from top to bottom would be described as `[1, 2, 3, 4]`.

- Certain elements on the *stack* need to be discussed frequently and have shorthand names in the documentation, interface, and code:
  - **bos** (Bottom of Stack) – the item listed at the bottom of the *Stack* window, `4` in the example above
  - **sos** (Second on Stack) – the item listed second from the bottom of the *Stack* window, `3` in the example above
  - **tos** (Top of Stack) – the item listed at the top of the *Stack* window, `1` in the example above

1.3 Concepts

When you start esc, you will see a status bar and four windows: *Stack*, *History*, *Commands*, and *Registers*. These windows match up neatly with the important concepts in esc.

Fig. 1: A typical esc window at startup.

1.3.1 Stack

The stack contains a series of numbers you are currently working with. Almost all commands follow the pattern of:

1. read numbers from the stack;
2. do something with them;
3. return other numbers to the stack.

As long as no other operation is in progress, your cursor will be positioned in the *Stack* window. To enter a new number (this is called *pushing it onto* the stack), simply start typing the number. The indicator at the left of the status bar will change to `[i]` to indicate that you're inserting a number, and the numbers will appear in the stack window as you type. When you're done typing the number, press `Enter` or `Space`; your cursor will then move to the next line of the stack. If you make a typo, correct it with `Backspace`. (If you've already pressed `Enter`, an *undo* will let you edit the number again.)

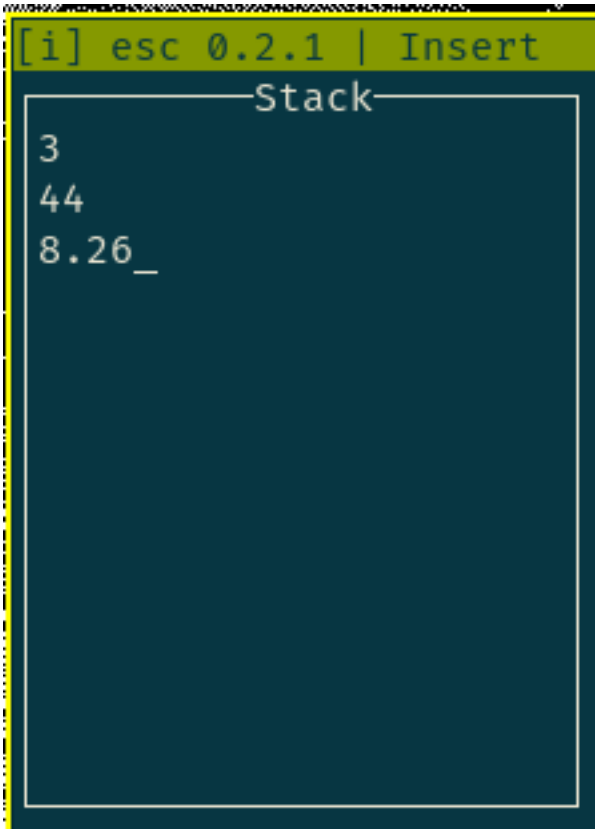


Fig. 2: Entering several numbers onto the stack.

You can enter as many numbers as you like this way, at least until the stack fills up. (In esc, the stack size is based on the number of lines allocated for the stack on the screen, which is ordinarily 12. 12 entries should be more than enough for all but the craziest calculations; many RPN-based HP calculators only have four, and that is normally enough.)

Enter negative signs with `_` (underscore) and scientific notation with `e` (e.g., `3.66e11` is 3.66 × 10^12 or 366,000,000,000).

Tip: You need only press `Enter` or `Space` between consecutively typed numbers. If you type a number and then wish to perform an *operation*, you can simply press the key for the operator. For instance, typing `2 2+` is equivalent

to typing `2 2 +`.

1.3.2 Commands

Arithmetic operations

Eventually you'll probably get tired of typing in numbers and want to actually perform some operation on the contents of the stack. In the *Commands* window, you will see a list of operations that you can perform. At the very top are listed the typical arithmetic operations. To perform one of these operations, simply press the associated key. For instance, to perform addition, you press +.

When you perform an operation, it removes a certain number of entries from the bottom of the stack (this is called *popping them off* the stack). It then uses those values to calculate the result and pushes the result back onto the bottom of the stack. For example, if your stack is `[1, 2, 3]` and you press the + key to add two numbers, the `1` is untouched, while the `2` and `3` are removed from the stack and the answer, `5`, is pushed on, so that the stack now has two entries, 1 and 5.

Note: Most operations pop one or two items from the stack and push one answer. However, this is not a requirement; an operator could pop four values and push two back.

You can see how many and what values an operation pops and pushes in its *help page*.

`esc.functions.`**`add`**(*sos*, *bos*)
Add sos and bos.

`esc.functions.`**`subtract`**(*sos*, *bos*)
Subtract bos from sos.

`esc.functions.`**`multiply`**(*sos*, *bos*)
Multiply sos and bos.

`esc.functions.`**`divide`**(*sos*, *bos*)
Divide sos by bos.

`esc.functions.`**`exponentiate`**(*sos*, *bos*)
Take sos to the power of bos.

`esc.functions.`**`modulus`**(*sos*, *bos*)
Take the remainder of sos divided by bos (a.k.a., sos mod bos).

`esc.functions.`**`sqrt`**(*bos*)
Take the square root of bos.

Stack operations

In addition to the arithmetic operations, some *stack operations* are provided. These don't calculate anything but allow you to manipulate the contents of the stack.

`esc.functions.`**`duplicate`**(*bos*)
Duplicate bos into a new stack entry. Useful if you want to hang onto the value for another calculation later.

`esc.functions.`**`exchange`**(*sos*, *bos*)
Swap bos and sos. Useful if you enter numbers in the wrong order or when you need to divide a more recent result by an older one.

esc.functions.**pop**(_)
Remove and discard the bottom item from the stack.

esc.functions.**roll**(**stack*)
Move the top item on the stack to the bottom.

esc.functions.**clear**(**stack*)
Clear all items from the stack, giving you a clean slate but maintaining your calculation history.

Command menus

Some entries in the *Commands* window don't immediately do anything but rather open a menu containing more commands, much like on a desktop scientific calculator. Simply choose an item from the menu to continue.

Other commands

In addition to the arithmetic and stack manipulation commands described above, esc defines several special commands.

esc.functions.**yank_bos**(*bos_str*, *testing*)
Copy the value of bos to your system clipboard.

builtin_stubs.py - stub classes for built-in commands

class esc.builtin_stubs.**StoreRegister**
Copy the bottommost value on the stack into a register. Registers store values under a single-letter name until you need them again.

See *Registers* for more information on registers.

class esc.builtin_stubs.**RetrieveRegister**
Copy the value of a register you've previously stored to the bottom of the stack. Registers store values under a single-letter name until you need them again.

See *Registers* for more information on registers.

class esc.builtin_stubs.**DeleteRegister**
Remove an existing register from your registers list and destroy its value.

See *Registers* for more information on registers.

class esc.builtin_stubs.**Undo**
Undo the last change made to your stack. Registers are unaffected. (This is a feature, not a bug: a common esc workflow is to reach an answer, then realize you need to go back and do something else with those same numbers. Registers allow you to hold onto your answer while you do so.)

See *History* for more information on calculation history.

class esc.builtin_stubs.**Redo**
Undo your last undo.

See *History* for more information on calculation history.

class esc.builtin_stubs.**Quit**
Quit esc. If you're in a menu, this option changes to "cancel" and gets you out of the menu instead.

Custom commands

In addition to all the commands described above, you may see some other commands in your list at times. These are added by *esc plugins*.

Getting help on commands

esc has a built-in help system you can use to discover what a command does, including the exact effect it would have on your current stack if you ran it. Simply press `F1`, then the key associated with the command. If you choose a *menu*, you'll get a description of the menu, but you can also choose an item from the menu to get specific help on that item.

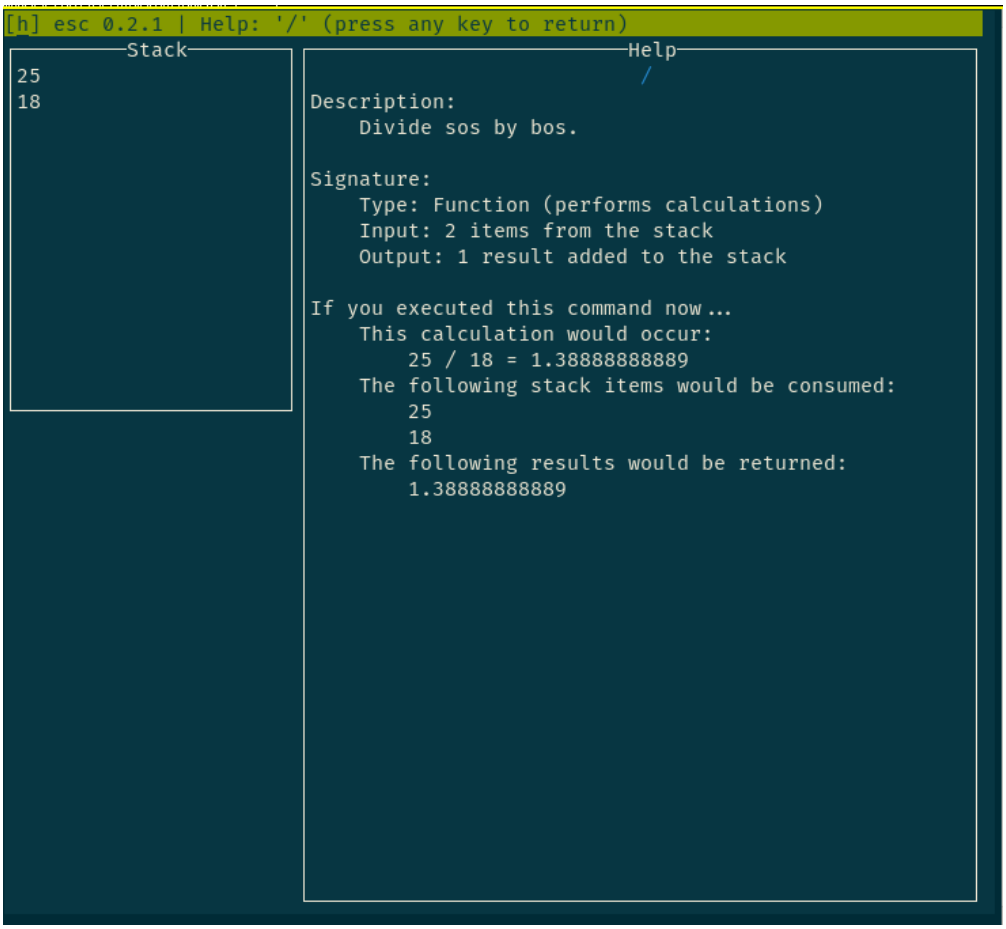


Fig. 3: Getting help on the division operation with several numbers on the stack. Press `F1 /` to reach this screen.

1.3.3 History

esc maintains a complete history of all the numbers you push onto the stack and operations you perform. The operations you execute and brief descriptions of their results are displayed in the *History* window in the middle of the screen.

If you perform an operation and then want to back up, simply choose `Undo`. To undo an undo, use `Redo`.

Calculation history you can step through is so useful it's amazing how few calculators offer it.

1.3.4 Registers

In addition to placing numbers on the stack, sometimes you might want to keep track of numbers in a slightly more permanent way. In this case, you can store the number to a register.

- To *store to a register*, press >, then type an upper- or lowercase letter to name the register. The bottom item on the stack is copied into the *Registers* window.
- To *retrieve the value of a register*, press <, then type the letter corresponding to the register whose value you want to retrieve. The value is copied into a new item at the bottom of a stack.
- To *delete a register*, press X, then type the letter of the register you want to delete. It is removed from the *Registers* window and its value is lost.

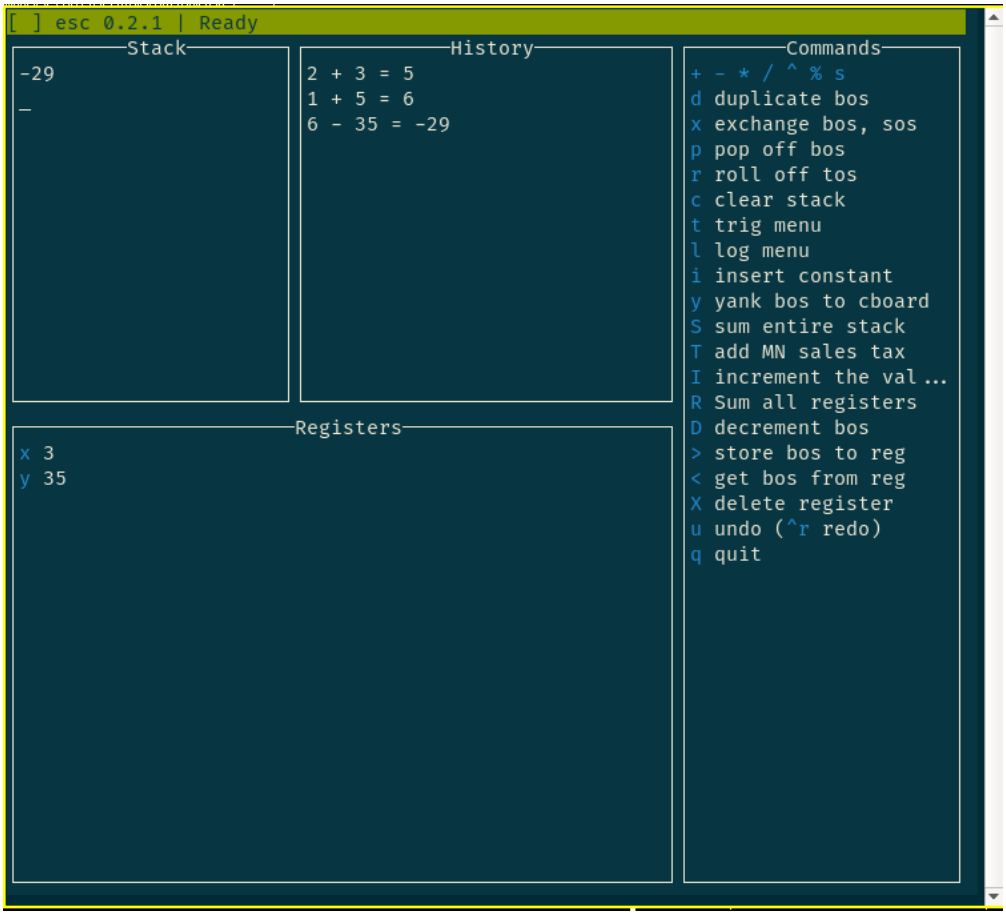


Fig. 4: Doing a few simple calculations, including placing some numbers in registers.

Note: Registers do not participate in the undo/redo history. This is a feature, not a bug: a common esc workflow is to perform some calculation, then realize you needed those numbers again for something. You can store your answer to a register, then undo as needed to get those numbers back.

1.4 Exercises

To get the hang of using esc and Reverse Polish Notation, here are a few simple exercises you can work through. Click the footnote link to show the answer.

1.4.1 Questions

Calculate the answers to the following algebraic expressions:

1. $2 + 5$[1]
2. $17/5$[2]
3. $\frac{30}{3 \cdot 5}$[3]
4. $60(2 \cdot (17 + 8))(5 \cdot 3)$[4]
5. $\frac{12+6}{18 \mod 3}$[5]
6. $\frac{1.52 \times 10^3}{12^{-2}}$[6]

Enter the numbers listed in the left column into the stack, then manipulate it using arithmetic or stack operations or registers to match the right column. You *cannot* enter any new numbers by typing them.

Answer	Initial stack	Final stack
[7]	5 3	3 5
[8]	1 2 3	(empty)
[9]	1 2 3	3 1 2
[10]	1 2 8	11 16 8

[1] **7** (`2 5 +`)
[2] **3.4** (`17 5 /`)
[3] **2** (`30 3 5 * /` or `3 5 * 30 x /`)
[4] **244800** (`60 2 17 8 + * * 5 3 * *` or `17 8 + 2 * 60 * 5 3 * *`)
[5] Trick question, this was a **division by zero error**! (`12 6 + 18 3 % /`)
[6] **218880** (`1.52e3 12 _2 ^ *`)

1.4.2 Answers

There are an infinite number of possible entry sequences that would work for every question. The sequence or sequences shown here are just some sensible choices; obviously, the most important thing is that you have the right answer.

In many cases, you'll have a choice between entering the numbers in strict order as they come in the expression or working from the inner parentheses out, or some combination thereof. This is largely a matter of taste, although in extremely large calculations you could run out of stack space if you type the numbers in order. This is not a serious threat in esc since the stack holds at least 12 numbers, but is more of a concern on hardware RPN calculators, which may hold as few as 3.

Note: Keystroke sequences are rendered with spaces between operations for readability here. Many of these spaces are not necessary when typing into esc or will even give the error "No number to finish adding".

1.5 Plugins

esc bundles only a small number of built-in operations so you can decide what operations will be useful to you and avoid staring all day at operations you never use. Any remaining operations you need can be added via plugins.

1.5.1 Plugin location

Plugins are Python modules (`.py` files) stored in the `plugins` subdirectory of your esc configuration directory. By default your esc configuration directory is `~/.esc`, where `~` represents your home or user directory. If this directory doesn't exist but `$XDG_CONFIG_HOME/esc` (or `~/.config/esc` if that environment variable is unset) does, that is used instead. (See here for why you might want to use the XDG folders specification.)

1.5.2 Installing plugins

To install a plugin, you simply copy its `.py` file into your plugins directory (see *Plugin location*). You can create the `.esc` and/or `plugins` directories if they don't exist. The next time you start esc, the plugins will be loaded.

> **Warning:** esc plugins can execute arbitrary Python code on your computer, which could include malicious code, so you should not install esc plugins from sources that you do not trust.

Tip: Plugins are loaded and their operations placed in the *Commands* window in alphabetical order by their filename. To control the order, you can prefix their filenames with numbers, e.g., `01_trig.py`, `02_log.py`.

1.5.3 Finding plugins

You can write your own plugins (see the *Developer Manual*) or get plugins from someone else. Some plugins providing common features like trig and log functions are available for download in the esc repository under esc-plugins.

[7] `x`
[8] `c`
[9] `r r`, or more playfully `x y p x ^V`, where ^V means to paste into esc using whatever method your terminal emulator requires!
[10] `>x + + <x d + <x`

CHAPTER 2

Developer Manual

The esc developer manual documents the API and internals of esc and describes how you add custom functionality and change the behavior of esc. If you're looking for information on installing or running esc or performing calculations, you want the *User Manual*.

If you haven't built an esc plugin before, start with the *Operations* section, which walks you through creating a plugin that implements a simple operation.

2.1 Operations

The most common thing to do in a plugin is to add a new operation. In the walkthrough presented on this page, we'll implement a new operation on the main menu. Our operation will calculate a proportion, like:

$$\frac{1}{2} = \frac{3}{x}$$

We'll pass the parameters in order, so when the stack reads `[1, 2, 3]`, we will obtain 6.

2.1.1 Creating a plugin

Create a new file in your *esc plugins directory* and paste in the following template:

```
"""
{name}.py - {description}
Copyright (c) 2019 {your name}.

{additional details}
"""

from esc.commands import Operation, main_menu
```

Change the sections in brackets to appropriate values for your plugins. Of course, the docstring is just a suggestion. Reload esc and ensure no errors occur. If you do get an exception, you probably made a syntax error; fix the file as necessary to resolve it.

2.1.2 Writing a function

Operations are written as Python functions decorated with `@Operation`. We'll start with the function and then look at the decorator.

How should we write this function? Let's generalize the example of a proportion calculation above, where d is the number we're solving for:

$$\frac{a}{b} = \frac{c}{d} \tag{2.1}$$

$$bc = ad \tag{2.2}$$

$$d = \frac{bc}{a} \tag{2.3}$$

(2.4)

We'll go ahead and use these letters for our variable names; they'll serve as well as anything else since this is a very general operation that could be used for just about anything. Translating the algebraic notation above into Python:

```
def proportion(a, b, c):
    return b * c / a
```

We can specify any number of parameters we want here and name them anything we want. When the user runs our operation, esc will check the function's parameter list to see how many parameters it has, slice that many items off the bottom of the stack, and bind them to the parameters in order. If there aren't enough items available, the user will get an error message telling them so. When we're done, we can return a single value or a tuple of values, and those values will replace the parameters that we received on the stack.

This is an oversimplification, as there are additional options that can change much of this behavior; we'll get to those in the discussion of the `@Operation` decorator.

Note: esc uses the Decimal library to implement decimal arithmetic similar to that of many handheld calculators. All function arguments are thus Decimal objects. Most operations on Decimals yield other Decimals, so you probably will not even notice if you're doing normal arithmetic on your arguments. If you ever get confused, check out the linked library documentation.

Return values from functions may be Decimal objects or any type that can be converted to a Decimal (string, integer, or float). Beware of returning floats except for numbers that are already irrational, as *all* the precision will be kept when converting back to the internal Decimal representation, even the rounding error inherent in binary floating-point values, which may result in silly values like `1.000000000083` appearing on the stack. If your function uses non-integer literals anywhere, it's a good idea to head this issue off by using the Decimal constructor to create them, like `from decimal import Decimal; x = Decimal("2.54")`.

2.1.3 Creating an @Operation

If you save the file and start esc, you won't get any errors, but you won't have any new operations either. In order to get an operation to show up, we need to add the `@Operation` decorator described earlier. That will make our code look like this:

```
@Operation(key='P', menu=main_menu, push=1,
           description="proportion from abc",
           log_as="{0}:{1} :: {2}:[{3}]")
def proportion(a, b, c):
    """
    Quickly calculate a proportion.
```

(continues on next page)

(continued from previous page)

```
    If the bottom three items on the stack are A, B, and C,
    then calculate D where A : B = C : D.
    """
    return b * c / a
```

Most of this is probably fairly self-explanatory, but a couple of points are worth noting.

- `log_as` is a format string whose positional placeholders will be replaced with a chain of the arguments to the function and the return values from the function. The formatted version will be used in the history window and help system.
- The function's docstring is used as the description in the help system.

`@Operation` can get more complicated, so without further ado here are the dirty details:

esc.commands.**Operation**(*key*, *menu*, *push*, *description=None*, *retain=False*, *log_as=None*, *simulate=True*)

Decorator to register a function on a menu and make it available for use as an esc operation.

Parameters

- **key** – The key on the keyboard to press to trigger this operation on the menu.
- **menu** – The *Menu* to place this operation on. The simplest choice is `main_menu`, which you can import from :mod:`esc.commands`.
- **push** – The number of items the decorated function will return to the stack on success. `0` means nothing is ever returned; `-1` means a variable number of things are returned.
- **description** – A very brief description of the operation this function implements, to be displayed next to it on the menu. If this is `None` (the default), the operation is "anonymous" and will be displayed at the top of the menu with just its *key*.
- **retain** – If `True`, the items bound to this function's arguments will remain on the stack on successful execution. The default is `False` (meaning the function's return value replaces whatever was there before – the usual behavior of an RPN calculator).
- **log_as** – A specification describing what appears in the *History* window after executing this function. It may be `None` (the default), `UNOP` or `BINOP`, a .format() string, or a callable.
  - If it is `None`, the *description* is used.
  - If it is the module constant `esc.commands.UNOP` or `esc.commands.BINOP`, the log string is a default suitable for many unary or binary operations: for `UNOP` it is *`description argument = return`* and for `BINOP` it is *`argument key argument = return`*.

    Note: If the function being decorated does not take one or two arguments, respectively, using `UNOP` or `BINOP` will raise a *`ProgrammingError`*.

  - If it is a format string, positional placeholders are replaced with the parameters to the function in sequence, then the return values. Thus, a function with two arguments `bos` and `sos` returning a tuple of two values replaces `{0}` with `bos`, `{1}` with `sos`, and `{2}` and `{3}` with the two return values.
  - If it is a callable, the parameters will be examined and bound by name to the following (none of these parameters are required, but arguments other than these will raise a *`ProgrammingError`*).

    args a list of the arguments the function requested

retval a list of values the function returned

registry the current *Registry* instance

The function should return an appropriate string.

- **simulate** – If `True` (the default), function execution will be simulated when the user looks at the help page for the function, so they can see what would happen to the stack if they actually chose the function. You should disable this option if your function is extremely slow or has side effects (e.g., changing the system clipboard, editing registers).

In addition to placing the function on the menu, the function is wrapped with the following magic.

1. Function parameters are bound according to the following rules:
   - Most parameters are bound to a slice of values at the bottom of the stack, by position. If the function has one parameter, it receives *bos*; if the function has two parameters, the first receives sos and the second bos; and so on. The parameters can have any names (see exceptions below). Using `bos` and `sos` is conventional for general operations, but if the operation is implementing some kind of formula, it may be more useful to name the parameters for their meaning in the formula.
   - By default, passed parameters are of type Decimal. If the parameter name ends with `_str`, it instead receives a string representation (this is exactly what shows up in the calculator window, so it's helpful when doing something display-oriented like copying to the clipboard). If the parameter name ends with `_stackitem`, it receives the complete *StackItem*, containing both of those representations and a few other things besides.
   - A varargs parameter, like `*args`, receives the entire contents of the stack as a tuple. This is invalid with any other parameters except `registry`. The `_str` and `_stackitem` suffixes still work. Again, it can have any name; `*stack` is conventional for esc operations.
   - The special parameter name `registry` receives a *Registry* instance containing the current state of all registers. Using this parameter is generally discouraged; see *Registry* for details.
   - The special parameter name `testing` receives a boolean describing whether the current execution is a test (see *esc.functest.TestCase()*). This can be useful if your function has side effects that you don't want to execute during testing, but you'd still like to test the rest of the function.
2. The function has a callable attached to it as an attribute, called `ensure`, which can be used to test the function at startup to ensure the function never stops calculating the correct answers due to updates or other issues:

```
def add(sos, bos):
    return sos + bos
add.ensure(before=[1, 2, 3], after=[1, 5])
```

   See *TestCase* for further information on this testing feature.

2.1.4 Writing tests

You probably don't want a calculator that returns the wrong results, so it's important to test your custom function! You could simply load esc and try it out, and that's a good idea regardless, but esc also offers built-in tests. These tests run automatically every time esc starts up; if they ever fail, esc will raise a *ProgrammingError* and refuse to load. This way, even if a new version of esc makes breaking changes you don't know about or you accidentally modify and break your function, you can be confident that esc won't return incorrect results (at least to the extent of your test coverage).

We can define automatic tests using the `ensure` attribute which the *@Operation* decorator adds to our function. Let's define a test that tests the example we discussed at the start of this page:

```
proportion.ensure(before=[1, 2, 3], after=[6])
```

Let's test an error condition too. What happens if calculating our proportion requires a divide by zero? Without special-casing that in our function, we would hope it informs the user that she can't divide by zero, which esc does by raising a `ZeroDivisionError` which is caught by the interface.

```
proportion.ensure(before=[0, 2, 3], raises=ZeroDivisionError)
```

And it's that easy. If you don't get a *ProgrammingError* after restarting esc, your tests pass.

Here's the full scoop on defining tests:

class esc.functest.**TestCase**(*before*, *after=None*, *raises=None*, *close=False*)

Test case defined with the `.ensure()` attribute of functions decorated with *@Operation*.

Parameters

- **before** – Required. A list of Decimals or values that can be coerced to Decimals. These values will be pushed onto a test stack that the operation will consume values from.
- **after** – Optional (either this or *raises* is required). A list of Decimals or values that can be coerced to Decimals. After the function is executed and changes the stack, the stack is compared to this list, and the test passes if they are identical.
- **raises** – Optional (either this or *after* is required). An exception type you expect the function to raise. This checks both the top-level exception type and any nested exceptions (since esc wraps many types of exceptions in *FunctionExecutionError*s). The test passes if executing the function (including all the machinery inside esc surrounding your actual function's execution, like pulling values off the stack) raises an exception of this type.
- **close** – If True, instead of doing an exact Decimal comparison, `math.isclose()` is used to perform a floating-point comparison. This is useful if dealing with irrational numbers or other sources of rounding error which may make it difficult to define the exact result in your test.

Test cases are executed every time esc starts (this is not a performance issue in practice unless you have a *lot* of plugins). If a test ever fails, a *ProgrammingError* is raised. Preventing the whole program from starting may sound extreme, but wrong calculations are pretty bad news!

Test cases are not inherently associated with an operation due to scope issues: Since the *EscOperation* itself is not returned to the functions file, only a decorated function, the function author can't access the *EscOperation*. Instead, they are associated with the function itself, and when the test() method is called on an *EscOperation*, it retrieves the test cases from the function and passes itself into the execute() method of each test case.

2.1.5 Putting it all together

Launch esc again. If you've made any mistakes, esc will hopefully catch them for you here and describe why you have an error or your test failed. Type in three values that can be used to calculate a proportion, choose the function from the menu, and you should be set!

2.1.6 Handling errors

esc handles many kinds of errors that could occur in your functions for you:

- If there aren't enough items on the stack to bind to all your arguments, your function won't even be called and the user will be told there aren't enough items on the stack.
- If your function raises a `ValueError`, the user will be informed a domain error has occurred (many math functions raise this exception in this case).
- If your function raises a `ZeroDivisionError`, the user will be informed he cannot divide by zero.
- If your function raises a Decimal `InvalidOperation`, the user will be informed the result is undefined. (The Decimal library in esc is configured so that `Infinity` is a valid result which occurs when extremely large numbers are put together such that the available precision is exceeded, but any result that would return `NaN` like `0 / 0` raises `InvalidOperation`.)

However, at times this will not be sufficient. One of the most common cases occurs when you need to work with the entire stack. In this case, you need to check yourself to see if there are sufficient items, as esc doesn't know how many you need. To do so, simply check the length of your `*args` tuple and raise an *InsufficientItemsError* if it's too short:

```
def my_operation(*stack):
    if len(stack) < 2:
        raise InsufficientItemsError(number_required=2)
    # do stuff
```

class esc.oops.**InsufficientItemsError**(*number_required*, *msg=None*)

Raised directly by operations functions that request the entire stack to indicate not enough items are on the stack to finish their work, or by the menu logic when an operation requests more parameters than items on the stack.

Functions may use the simplified form of the exception, providing an int describing the number of items that should have been on the stack for the `number_required` constructor parameter. esc will then reraise the exception with a more useful message; a fallback message is provided in case this doesn't happen for some reason.

Another case arises when your function encounters some arbitrary condition that prevents it from continuing. As a silly example, perhaps the result of your formula is undefined if the sum of its input values is 6. Raising a *FunctionExecutionError* with a message argument will cause the function's execution to stop and the message to be printed to the status bar. (The stack will remain unchanged.) As noted below, the message should be concise so it fits in the status bar – it will be truncated if it doesn't fit.

```
def my_operation(sos, bos):
    if sos + bos == 6:
        raise FunctionExecutionError("I don't like the number 6.")
    # do stuff
```

class esc.oops.**FunctionExecutionError**

A broad exception type that occurs when the code within an operation function couldn't be executed successfully for some reason. Examples include:

- a number is in the middle of being entered and isn't a valid number
- a function performed an undefined operation like dividing by zero
- there are too many or too few items on the stack
- a function directly raises this error due to invalid input or inability to complete its task for some other reason

*FunctionExecutionError*s normally result in the `__str__` of the exception being printed to the status bar, so exception messages should be concise.

A *FunctionExecutionError* may be raised directly, or one of its subclasses may be used.

Warning: If you do something complicated in your function that could result in an exception other than the types listed above, be aware that if you let an exception of another type bubble up from your function, esc will crash and show the traceback. This is generally reasonable behavior if you don't expect the error, since it makes it easy to spot and fix the problem, but if the error is an expected possibility you'll probably want to catch it and give the user a helpful error message describing the problem by raising a *FunctionExecutionError*.

2.2 Menus

Many *operations* can go on the main menu, but at some point you may want to create additional menus or register new items on existing menus.

Menus can have submenus to an effectively infinite depth.

2.2.1 Creating new menus

Menus are defined with the *Menu* constructor:

esc.commands.**Menu**(*key*, *description*, *parent*, *doc*, *mode_display=None*)
Register a new submenu of an existing menu.

Parameters

- **key** – The keyboard key used to select this menu from its parent.
- **description** – A short description of this menu to show beside the key.
- **parent** – An *EscMenu* to add this menu to. This may be esc.commands.main_menu or another menu.
- **doc** – A string describing the menu, to be used in the help system. This should be something like the docstring of an operation function.
- **mode_display** – An optional callable returning a string whose value will be shown beneath the name of the menu when the menu is open. Ordinarily, this is used to show the current value of any modes that apply to the functions on the menu.

Returns A new *EscMenu*.

Example:

```
from esc.commands import Operation, Menu, main_menu

qdoc = """
This menu contains common operations needed when working with
problems in queueing theory.
"""
qmenu = Menu('Q', 'queueing menu', parent=main_menu, doc=qdoc)

# Here we would define functions whose @Operation decorators
# take 'qmenu' as their menu= argument.
```

2.2.2 Adding to existing menus

Sometimes it's useful to extend menus defined by other plugins. This poses a challenge: how do we get access to those menu objects? The plugins directory is not a package and we can't guarantee its contents, so it's tricky to import from

other plugins. The easiest method uses the *esc.commands.EscMenu.child()* method:

EscMenu.**child**(*access_key*)
Return the child defined by *access_key*. Raises *NotInMenuError* if it doesn't exist.

We only need to know the menu access keys to get at any item from the main menu! Let's suppose we want to add a `secant` operation to the trig menu installed by the *trig* plugin bundled with esc. In order to do that, we need to obtain the trig menu. The key of this menu is `t`. Thus we would do:

```
from esc.commands import main_menu

trig_menu = main_menu.child('t')
```

This isn't very robust, though. We probably want to make sure we've actually gotten the trig menu and not some other menu that happened to have the access key `t`, and we might want our plugin to add its own operations even if we don't have the trig-menu plugin installed. Let's add these features:

```
from esc.commands import main_menu
from esc.oops import NotInMenuError, ProgrammingError

if 't' in main_menu.children:
    trig_menu = main_menu.child('t')
    if trig_menu.description != 'trig menu':
        raise ProgrammingError(
            f"Expected the menu 't' to be the trig menu, but it was "
            f"{trig_menu.description}.")
else:
    trig_doc = """
        Calculate the values of trigonometric functions, treating inputs as
        either degrees or radians depending on the mode.
    """
    trig_menu = Menu('t', 'trig menu', parent=main_menu,
                     doc=trig_doc,
                     mode_display=trig_mode_display)
```

This is better. If the menu doesn't exist already, we'll create it; if it does, we'll retrieve it and then be sure it's the right one, throwing an error if it isn't.

Warning: It's important to make sure your plugin loads *after* the plugin you're extending, or that plugin will crash when it tries to add a menu that already exists (unless it does the same check you did). That's easy enough to do by *changing the filename* to be alphabetically later than the plugin you're targeting; an easy way is to name your file as the target plugin with an additional suffix, like `trigextensions.py`. (Don't use an underscore, though – that sorts before the dot!)

Note: A good future addition to esc would be a convenient way to remap the keys added by different plugins, since it's easy for them to end up colliding if they aren't designed with each other in mind. As of now, this is a bit hacky still and manual modification is required if you encounter a collision.

2.3 Constants

Adding new constants is easy as pi with the *Constant* constructor:

esc.commands.**Constant**(*value*, *key*, *description*, *menu*)
Register a new constant. Constants are just exceedingly boring operations that pop no values and push a constant value, so this is merely syntactic sugar.

Parameters

- **value** – The value of the constant, as a Decimal or a value that can be converted to one.
- **key** – The key to press to select the constant from the menu.
- **description** – A brief description to show next to the *key*.
- **menu** – A *Menu* to place this function on.

The one trick here is that if you want to add your constant to the constants menu, you may not have access to the constants menu. In this case, use the trick for *adding to existing menus*:

```
from esc.commands import Constant, main_menu
constants_menu = main_menu.child('i')
Constant(299792458, 'c', 'speed of light (m/s)', constants_menu)
```

Since the constants menu is built in to esc, `i` will always be the constants menu and there is no need to perform the other checks described in the linked section.

2.4 Modes

Modes are global state that can be used to allow esc operations to work in different ways. For instance, the trig operations in the *trig* plugin distributed with esc use a degrees/radians mode. Modes tend to be confusing to both users and programmers, so it's best to avoid them when possible, but sometimes it's difficult to do without them (do you really want to create and select a whole separate set of trig operations depending on whether you're calculating in degrees or radians?).

2.4.1 Working with modes

Modes are created using the *Mode* and *ModeChange* constructors:

esc.commands.**Mode**(*name*, *default_value*, *allowable_values=None*)
Register a new mode.

Parameters

- **name** – The name of the mode. This is used to refer to it in code. If a mode with this name already exists, a *ProgrammingError* will be raised.
- **default_value** – The value the mode starts at.
- **allowable_values** – An optional sequence of possible values for the mode. If defined, if code ever tries to set a different value, a *ProgrammingError* will be raised.

esc.commands.**ModeChange**(*key*, *description*, *menu*, *mode_name*, *to_value*)
Create a new mode change operation the user can select from a menu. Syntactic sugar for registering an operation.

Parameters

- **key** – The key to press to select the constant from the menu.
- **description** – A brief description to show next to the *key*.
- **menu** – A *Menu* to place this operation on.

- **mode_name** – The name of the mode, registered with *Mode()*, to set.
- **to_value** – The value the mode will be set to when this operation is selected.

Aside from defining *ModeChange* operations, you can view and edit the values of modes using the *modes* module:

modes.py - Manage calculator state/modes

class esc.modes.**Mode**(*name*, *value*, *allowable_values*)
esc modes implement basic calculator state like a degrees/radians switch. In esc, they are created and used by menus with families of related operations, where they can also be displayed. They have a name, a current value, and optionally a set of allowable values; if something ever causes the value to be set to a non-allowable value, a *ProgrammingError* will be raised, hopefully identifying the issue before it leads to wrong results.

Modes are usually created by the *esc.commands.Mode()* factory function, not by calling this constructor directly.

esc.modes.**get**(*name*)
Retrieve the value of a mode with a given name. Return None if no mode by that name has been registered.

esc.modes.**register**(*name*, *default_value*, *allowable_values=None*)
Create a new mode. If the mode already exists, a *ProgrammingError* is raised.

Modes should be registered by the *esc.commands.Mode()* factory function, not by calling this function directly.

esc.modes.**set**(*name*, *val*)
Set a mode to a new value. If the mode doesn't exist, a KeyError will be raised. If the value is invalid for the mode, a *ProgrammingError* will be raised.

You'll probably want to display the value of your mode somewhere – as confusing as modes can be already, they're even worse when they're invisible! Typically, this is done by supplying a mode_display callable to the *Menu* that contains the associated operations. This callable takes no arguments and calls *modes.get* to determine the display value:

```
def my_mode_display_1():
    # Assumes the values of the mode are strings.
    # If they're something else, you need to map them to strings here.
    return modes.get('my_mode')
```

2.4.2 Example

Here's a complete silly example:

```
from esc.commands import Operation, Mode, ModeChange, Menu, main_menu, BINOP
from esc import modes

def opposite_mode():
    return "[opposite day]" if modes.get('opposite_day') else ""
bool_menu = Menu('b', 'boolean functions', parent=main_menu, doc="blah",
                 mode_display=opposite_mode)

Mode('opposite_day', False, (True, False))
ModeChange("o", "enable opposite day", menu=bool_menu,
           mode_name='opposite_day', to_value=True)
ModeChange("O", "disable opposite day", menu=bool_menu,
           mode_name='opposite_day', to_value=False)

@Operation('a', menu=bool_menu, push=1,
```

(continues on next page)

(continued from previous page)

```
            description="AND sos and bos",
            log_as=BINOP)
def and_(x, y):
    result = x and y
    return int((not result) if modes.get('opposite_day') else result)

# ...insert other boolean functions here
```

2.5 Debugging

If you do something wrong in a plugin, a *ProgrammingError* will be raised:

class esc.oops.**ProgrammingError**

Indicates an error caused by incorrectly written or defined esc plugins. This includes modes, menus, operations, and so on. It does not include runtime errors within operation functions themselves; these are *FunctionExecutionError*s.

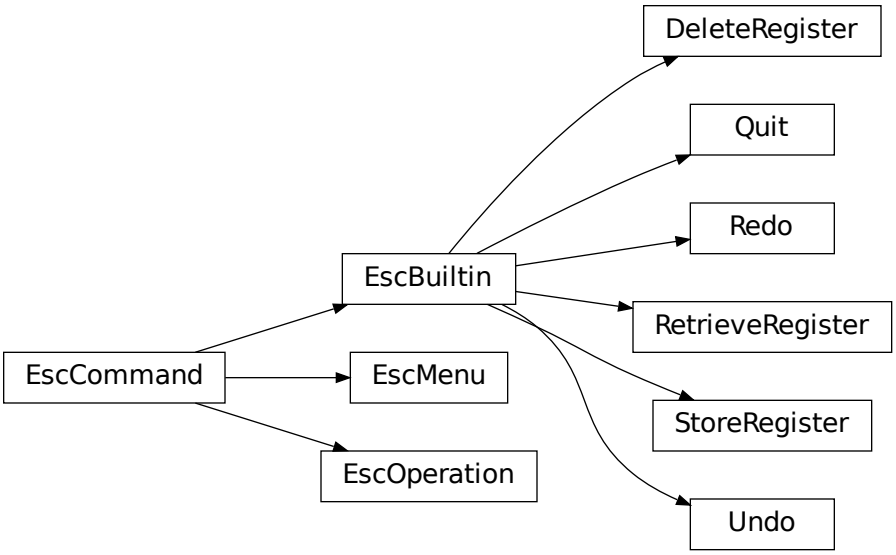
The traceback will usually contain enough information to identify the problem. If you have difficulty with the operation function itself (rather than interacting with the esc framework), you may wish to pull the function out into a test script or use the `logging` standard library module, as it can be quite challenging to debug a function within the context of a curses app without a separate connected debugger (the terminal is taken over, precluding the use of `print()` statements or a direct drop into `pdb`). One quick-and-dirty hack is to raise an `Exception` with a value you want to check as the message; this will only let you check one value per run, but it can still be useful.

2.6 Class Reference

This page describes some miscellaneous classes you may encounter while developing esc plugins and want to know more about. This is not an exhaustive reference to all of esc's classes; for details on classes that plugin developers do not come into contact with, you'll want to dive into the source.

2.6.1 EscCommands

Anything that shows up on the *Commands* menu has an associated instance of *EscCommand*. *EscCommand* is subclassed as follows:



EscCommand subclasses

class esc.commands.**EscCommand**(*key*, *description*)

Base class for some esc functionality or operation the user can activate.

When the user activates this item, *execute()* is called. Execution takes any action associated with the item, throwing an exception if something didn't work right. It then returns the menu that the interface should return to. A return value of None returns to the main menu.

children = None

Mapping from keys to *EscCommand*s on the current menu, if this is a menu. Add to this using *register_child()*, not directly.

description = None

How this item is described on its parent menu.

execute(*access_key*, *ss*, *registry*)

Execute this EscCommand. For operations or builtins, this involves the class doing its own work; for menus, this returns the child defined by *access_key*.

Returns An instance of *EscCommand* representing the menu the UI should now return to, or None to indicate the main menu.

help_title

The title this command should show in the help system. This is the access key and description if a description is defined; otherwise it is just the access key.

key = None

The key used to activate this item on its parent menu.

parent = None

An *EscCommand* (hopefully a menu) this item is contained in.

register_child(*child*)
Register a new child *EscCommand* of this menu (either a menu or an operation). This operation doesn't make sense for *EscOperation* instances; the caller should avoid doing this.

signature_info
An iterable of strings to display under the "Signature" section in help.

simulated_result(*ss*, *registry*)
Execute this command against the given stack state and registry, but instead of actually changing the state, return a string describing the result.

May return `None` if the *EscCommand* does not change the stack state (e.g., a menu).

test()
Execute any self-tests associated with this *EscCommand*. If a test fails, raise a *ProgrammingError*.

class esc.commands.**EscMenu**(*key*, *description*, *doc*, *mode_display=None*)
Bases: *esc.commands.EscCommand*

A type of EscCommand that serves as a container for other menus and operations. Executing it activates a child item.

anonymous_children
Iterable of children without a description.

child(*access_key*)
Return the child defined by *access_key*. Raises *NotInMenuError* if it doesn't exist.

execute(*access_key*, *ss*, *registry*)
Look up the child described by *access_key* and execute it. If said child is a menu, return it (so the user can choose an item from that menu). Otherwise, execute the child immediately.

Parameters

- **access_key** – A menu access key indicating which child to execute.
- **ss** – The current stack state, passed through to a child operation.
- **registry** – The current registry, passed through to a child operation.

Returns The *EscMenu* to display next, or `None` to return to the main menu. This will be a child menu, if one was selected, or None if an operation runs.

Raises *FunctionExecutionError* or a subclass, if a child operation was selected but does not complete successfully.

If the user chose the special quit command, return to the previous menu, or raise `SystemExit` if this is the main menu.

is_main_menu
This is the main menu if it has no parent.

mode_display = None
An optional callable whose return value will be shown under the menu title.

named_children
Iterable of children with a description.

signature_info
Constant string that describes the menu as a menu.

test()
Execute the test method of all children.

class esc.commands.**EscOperation**(*key*, *func*, *pop*, *push*, *description*, *menu*, *retain=False*, *log_as=None*, *simulate=True*)

Bases: *esc.commands.EscCommand*

execute(*access_key*, *ss*, *registry*)

Execute the esc operation wrapped by this instance on the given stack state and registry.

Parameters

- **access_key** – Not used by this subclass.
- **ss** – The current stack state, passed through to a child operation.
- **registry** – The current registry, passed through to a child operation.

Returns A constant `None`, indicating that we go back to the main menu.

Raises *FunctionExecutionError* or a subclass, if the operation cannot be completed successfully.

function = None

The function, decorated with *@Operation*, that defines the logic of this operation.

log_as = None

A description of how to log this function's execution (see the docs for *@Operation* for details on allowable values).

pop = None

The number of items the function gets from the bottom of the stack. `-1` indicates the entire stack is popped.

push = None

The number of items the function returns to the stack. `-1` indicates a variable number of items will be returned.

retain = None

If true, items pulled from the stack before execution won't be removed.

signature_info

A description of the function's signature as a tuple of strings (one per line to display in the help system), based on the *pop* and *push* values.

simulate_allowed = None

Whether this function should be run when a simulation is requested for help purposes. Turn off if the function is slow or has side effects.

simulated_result(*ss*, *registry*)

Execute the operation on the provided *StackState*, but don't actually change the state – instead, provide a description of what would happen.

test()

If the function on this *EscOperation* has associated *TestCase*s defined in its *tests* attribute, execute those tests.

class esc.commands.**EscBuiltin**

Bases: *esc.commands.EscCommand*

Mock class for built-in commands. Built-in *EscCommand*s do not actually get run and do anything – they are special-cased because they need access to internals normal commands cannot access. However, it's still useful to have classes for them as stand-ins for things like retrieving help.

Unlike the other *EscCommand*s, each *EscBuiltin* has its own subclass rather than its own instance, as they each need special behaviors. The subclasses are defined in the *builtin_stubs* module.

Subclasses should override the docstring and the *simulated_result()* method.

Subclasses should also define *key* and *description* as class variables. They'll be shadowed by instance variables once we instantiate the class, but the values will be the same. That sounds dumb, but it makes sense for all other classes in the hierarchy and doesn't hurt us here. We don't want to define them in the __init__ of each subclass because then we have to instantiate every class to match on them by key (see the reflective search in esc.helpme).

execute(*access_key*, *ss*, *registry*)
Executing a builtin does nothing.

signature_info
Constant string that describes the built-in as a built-in.

simulated_result(*ss*, *registry*)
Reimplemented by each subclass.

test()
Testing a builtin with esc's function test feature does nothing.

Loading EscCommands

*EscCommand*s are loaded by the *function_loader*:

function_loader.py - load esc functions from builtins and plugins onto menus

esc.function_loader.**load_all**()
Load built-in and user functions files. This will execute the constructors in the functions files, which will (if these files are written correctly) ultimately register the functions onto main_menu. This method needs to be called only once at application startup.

The function loader imports the built-in functions file and any function files in your *user config directory*. Importing a function file causes calls to the constructors in *EscCommand* to be run (*Operation()*, *Menu()*, *Constant()*, *Mode()*, and *ModeChange()*), and these constructors in turn create *EscCommand* instances which are added to the *children* attribute of the main menu or a submenu of the main menu.

2.6.2 StackItems

*StackItem*s are used to represent each number entered onto the stack. Typically, you can request a Decimal or string representation of the stack item in your operation functions (see the documentation on parameter binding in *Operation* for details). If you need both in one function, you may want to work with the full object:

class esc.stack.**StackItem**(*firstchar=None*, *decval=None*)
An item placed on esc's stack. At its root, this is a number, but it gets more complicated than that!

For one, we need a numeric value for calculations as well as a string value to display on the screen. The method *finish_entry()* updates the numeric representation from the string representation. We could dynamically compute the string representation with reasonable performance, but see the next paragraph for why this isn't helpful.

For another, a stack item may be *incomplete* (*is_entered* attribute = False). That's because the user doesn't enter a number all at once – it will typically consist of multiple keystrokes. If so, there won't be a decimal representation until we call *finish_entry()*. The StackState is in charge of calling this method if needed before trying to do any calculations with the number.

Note: By the time you receive a *StackItem* in a plugin function, *is_entered* should always be `False`, so many of the following methods will not be applicable.

add_character(*nextchar*)
Add a character to the running string of the number being entered on the stack. Calling *add_character()* is illegal and will raise an `AssertionError` if the number has already been entered completely.

Returns `True` if successful, `False` if the stack width (`esc.consts.STACKWIDTH`) has been exceeded.

backspace(*num_chars=1*)
Remove the last character(s) from the string being entered. Calling *backspace()* is illegal and will raise an `AssertionError` if the number has already been entered completely.

decimal = None
Decimal representation. This is `None` if *is_entered* is `False`.

finish_entry()
Signal that the user is done entering a string and it should be converted to a Decimal value. If successful, return `True`; if the entered string does not form a valid number, return `False`. This should be called only by the `enter_number` method of `StackState`.

is_entered = None
Whether the number has been fully entered. If not entered, many methods will not work as we don't have a Decimal representation yet.

string = None
String representation.

2.6.3 Registry

In general, you should prefer working solely with the stack when writing operations, rather than accessing *registers*; operations that use registers are harder to write and harder for users to understand, and any changes made to registers can't be undone. (Working through the undo/redo history will still deliver the correct values to the stack, since undoing and redoing restores past states rather than doing the calculations again, but history entries might not match the current values of registers anymore, and changes your operation makes to registers won't be undone/redone.)

However, sometimes it may be useful to use registers as parameters or even as outputs in special-purpose custom operations. *You should do this only if you fully understand the consequences as described in the previous paragraph!* In this case, you can provide a parameter called `registry`, and you will receive the following object:

class esc.registers.**Registry**
The Registry stores the values of esc registers. It's basically a fancy dictionary with some validation and a display-sorted *items()* method.

__contains__(*value*)

__delitem__(*key*)

__getitem__(*key*)

__len__()

__setitem__(*key*, *value*)
Set the value of a register.

Raises *InvalidNameError* if the key (register name) isn't valid.

static _valid_name(*name: str*)
A key (register name) is valid if it's exactly one alphabetic character.

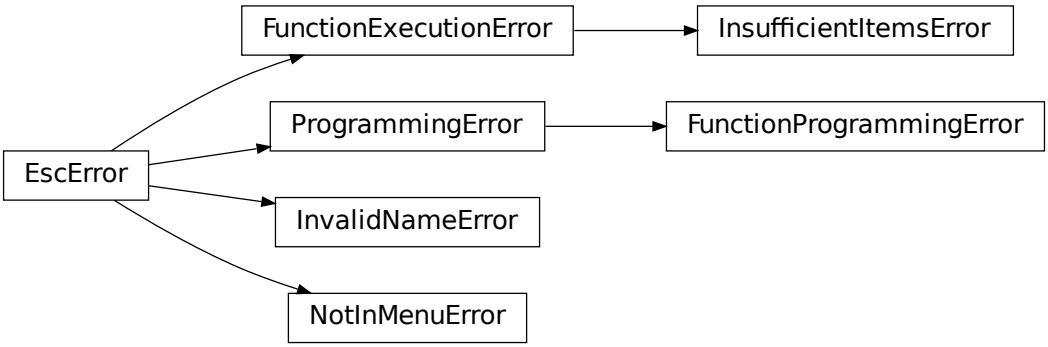
items()
Return an iterable of items, sorted by register key.

values()

Warning: If you *set* a register in your operation, be sure to turn the `simulate <esc.commands.Operation.simulate>` option off in your *Operation* decorator, or users may end up inadvertently setting registers when viewing the help for your function.

2.6.4 Exceptions

The most important exceptions are described (and indexed) in the context where you need to deal with them, but here is the complete hierarchy for reference.



class esc.oops.**EscError**
Base application exception for esc. Don't raise directly.

class esc.oops.**FunctionExecutionError**
A broad exception type that occurs when the code within an operation function couldn't be executed successfully for some reason. Examples include:

- a number is in the middle of being entered and isn't a valid number
- a function performed an undefined operation like dividing by zero
- there are too many or too few items on the stack
- a function directly raises this error due to invalid input or inability to complete its task for some other reason

*FunctionExecutionError*s normally result in the `__str__` of the exception being printed to the status bar, so exception messages should be concise.

A *FunctionExecutionError* may be raised directly, or one of its subclasses may be used.

class esc.oops.**InsufficientItemsError**(*number_required*, *msg=None*)
Raised directly by operations functions that request the entire stack to indicate not enough items are on the stack to finish their work, or by the menu logic when an operation requests more parameters than items on the stack.

Functions may use the simplified form of the exception, providing an int describing the number of items that should have been on the stack for the `number_required` constructor parameter. esc will then reraise the exception with a more useful message; a fallback message is provided in case this doesn't happen for some reason.

class esc.oops.**ProgrammingError**
Indicates an error caused by incorrectly written or defined esc plugins. This includes modes, menus, operations, and so on. It does not include runtime errors within operation functions themselves; these are *FunctionExecutionError*s.

class esc.oops.**FunctionProgrammingError**(*operation*, *problem*)
A more specific type of *ProgrammingError* that occurs when a user's *@Operation* decorator or function parameters are invalid or function tests fail.

The distinction is mostly for convenience within esc's codebase rather than because client code needs to tell the difference from other *ProgrammingError*s; this class wraps some handy logic for generating a standardized message.

class esc.oops.**InvalidNameError**
Raised when the user chooses an invalid name for a register or other label.

class esc.oops.**NotInMenuError**(*access_key*)
Raised when a keypress is parsed as a menu option or a menu's children are programmatically accessed by key, but the key doesn't refer to any choice on the current menu. Describes the problem in a message that can be printed to the status bar.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M