Pfp (python format parser) is a python interpreter for 010 Editor template scripts.

Pfp uses py010parser to parse 010 templates into an AST, which is then interpreted by pfp. Pfp then returns a DOM object which can be used to access individual fields of the defined data structure.

Please read the *Getting Started* section for a better introduction.
1.1 Installation

pip install pfp

1.2 Console Script

Pfp comes with a console script that will print parsed data:

```bash
$> pfp --help
usage: pfp [-h] -t TEMPLATE [--show-offsets] [-k] input

Run pfp on input data using a specified 010 Editor template for parsing

positional arguments:
  input The input data stream or file to parse. Use '-' for piped data

optional arguments:
  -h, --help show this help message and exit
  -t TEMPLATE, --template TEMPLATE The template to parse with
  --show-offsets Show offsets in the parsed data of parsed fields
  -k, --keep Keep successfully parsed data on error
```

Example usages:

```bash
pfp --keep -t png.bt test.png

cat test.png | pfp --keep -t png.bt -
pfp --keep -t png.bt - <test.png
```
1.3 PNG Parsing Example

Below is a simple PNG template that will parse the PNG image into chunks. The tEXt chunk of the PNG image will also specifically be parsed:

```c
typedef struct {
    // null-terminated
    string label;
    char comment[length - sizeof(label)];
} TEXT;

typedef struct {
    uint length<watch=\text{data}, update=WatchLength>;
    char cname[4];
    union {
        char raw[length];
        if (cname == "tEXt") {
            TEXT tEXt;
        }
    } data;
    uint32 crc<watch=cname;data, update=WatchCrc32>;
} CHUNK;

uint64 magic;

while(!FEof()) {
    CHUNK chunks;
}
```

The python code below will use the template above to parse a PNG image, find the tEXt chunk, and change the comment:

```python
import pfp
dom = pfp.parse(data_file="image.png", template_file="png_template.bt")

for chunk in png.chunks:
    if chunk.cname == "tEXt":
        print("Comment before: ").format(chunk.data.tEXt.comment)
        chunk.data.tEXt.comment = "NEW COMMENT"
        print("Comment after: ").format(chunk.data.tEXt.comment)
```
A few differences do exist between 010 Editor and pfp. See the *Differences Between 010 and pfp* section for specific, documented differences.

Contents:

2.1 Getting Started

2.1.1 Installation

Pfp can be installed via pip:

```
pip install pfp
```

2.1.2 Introduction

Pfp is an interpreter for 010 template scripts. 010 Template scripts use a modified C syntax. Control-flow statements are allowed within struct declarations, and type checking is done dynamically, as statements are interpreted instead of at compile time.

010 template scripts parse data from the input stream by declaring variables. Each time a variable is declared, that much data is read from the input stream and stored in the variable.

Variables are also allowed that do not cause data to be read from the input stream. Prefixing a declaration with `const` or `local` will create a temporary variable that can be used in the script.

An example template script that parses TLV (type-length-value) structures out of the input stream is shown below:

```
local int count = 0;
const uint64 MAGIC = 0xaabbccddeeff0011;

uint64 magic;
```
if (magic != MAGIC) {
    printf("Magic value is not valid, bailing");
    return 1;
}

while (!FEof()) {
    printf("Parsing the %d-th TLV structure", ++count);
    struct {
        string type;
        int length;
        char value[length];
    } tlvs;
    }
}

Note that a return statement in the main body of the script will cause the template to stop being executed. Also note that declaring multiple variables of the same name (in this case, tlvs) will cause that variable to be made into an array of the variable’s type.

More about the 010 template script syntax can be read about on the 010 Editor website.

2.1.3 Parsing Data

010 template scripts are interpreted from python using the `pfp.parse` function, as shown below:

```python
import pfp

template = ""
    local int count = 0;
    const uint64 MAGIC = 0xaabbccddeeff0011;
    uint64 magic;

    if (magic != MAGIC) {
        printf("Magic value is not valid, bailing");
        return 1;
    }

    while (!FEof()) {
        printf("Parsing the %d-th TLV structure", ++count);
        struct {
            string type;
            int length;
            char value[length];
        } tlvs;
        }
    ""

parsed_tlv = pfp.parse(
    template = template,
    data_file = "path/to/tlv.bin"
)
```

The `pfp.parse` function returns a dom of the parsed data. Individual fields may be accessed using standard dot-notation:
```
for tlv in parsed_tlv.tlvs:
    print("type: {}, value: ")\n        .format(tlv.type, tlv.value))
```

### 2.1.4 Manipulating Data

Parsed data contained within the dom can be manipulated and then rebuilt:

```
for tlv in parsed_tlv.tlvs:
    if tlv.type == "SOMETYPE":
        tlv.value = "a new value"
new_data = parsed_tlv._pfp__build()
```

### 2.1.5 Printing Structures

The method `pfp.fields.Field._pfp__show` will print data information about the field. If called on a field that contains child fields, those fields will also be printed:

```
dom = pfp.parse(...)\nprint(dom._pfp__show(include_offset=True))
```

### 2.1.6 Metadata

010 template syntax supports adding “special attributes” (called metadata in pfp). 010 editor’s special attributes are largely centered around how fields are displayed in the GUI; for this reason, pfp currently ignores 010 editor’s special attributes.

However, pfp also introduces new special attributes to help manage relationships between fields, such as lengths, checksums, and compressed data.

The template below has updated the TLV-parsing template from above to add metadata to the length field:

```
local int count = 0;
const uint64 MAGIC = 0xaabbccddeeff0011;
uint64 magic;

if(magic != MAGIC) {
    printf("Magic value is not valid, bailing");
    return 1;
}
while(!Feof()) {
    printf("Parsing the %d-th TLV structure", ++count);
    struct {
        string type;
        int length<watch=value, update=WatchLength>;
        char value[length];
    } tlv;
}
```

With the metadata, if the `value` field of a `tlv` were changed, the `length` field would be automatically updated to the new length of the `value` field.

### 2.1. Getting Started
2.1.7 Debugger

Pfp comes with a built-in debugger, which can be dropped into by calling the `Int3()` function in a template.

```cpp
23 // length (4 bytes), chunk_type (4 bytes), data (length bytes), crc (4 bytes)
24 // CRC Does NOT include the length bytes.
25 //--------------------------------------
26
--> 27 Int3();
28
29 BigEndian(); // PNG files are in Network Byte order
30
31 const uint64 PNGMAGIC = 0x89504E470D0A1A0AL;
```

```
pfp> peek
89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52 .PNG........IHDR
pfp> help
Documented commands (type help <topic>):
========================================
EOF continue eval help list next peek quit s show step x
```

```
pfp> n
25 //--------------------------------------
26
27 Int3();
28
--> 29 BigEndian(); // PNG files are in Network Byte order
30
31 const uint64 PNGMAGIC = 0x89504E470D0A1A0AL;
```

2.2 Metadata

Fields in PFP are allowed to have metadata. Metadata is added to a field by adding `key=val,key2=val2,..` after a field’s declaration, but before the semicolon. 010 templates also allow for metadata to be added to fields, although most of those values changed how fields were displayed in the GUI:

```cpp
int someField<format=hex>;
```

PFP adds some more useful extensions to the 010 template syntax. E.g. metadata values that allow fields to “watch” a different field and update its own value when the watched field changes:

```cpp
struct {
    int length<watch=stringData, update=WatchLength>;
    string data;
} stringWithLength;
```
2.2.1 PFP Metadata Extensions

Watch Metadata

Watch metadata allows the template to specify that a field should be modified or update when one of the fields it
watches changes value.

Watch metadata must meet the requirements below:

- must contain the *watch* key to specify which field(s) to watch
- must contain the *update* key to specify a function to perform the updating

**watch**

The watch key must be one or more semi-colon-separated statements or field names. All of the these fields will be
passed to the specified *update* function. E.g.:

```
int field1;
int field2;
int field3<watch=field1;field2, ...>;
```

Note that each item in the semi-colon-separated watch field list is eval’d as 010 template script. The resulting field
will be the result of the eval. This allows, for example, functions to be called that will return which field to watch. (I
have no idea why you’d want to do this, but you can).

**update**

The update key must be the name of a function, native or interpreted, that will accept at least two parameters. The
update function should have the signature:

```
void SumFields(int &to_update, int watched1, int watched2) {
    to_update = watched1 + watched2;
}
```

The function above can then be used like so:::

```
int field1; int field2; int sum<watch=field1;field2, update=SumFields>;
```

**Built-in Watch Functions**

pfp.native.watchers.watch_crc(*args, **kwargs)

WatchCrc32 - Watch the total crc32 of the params.

**Example:** The code below uses the WatchCrc32 update function to update the crc field to the crc of the
length and data fields

```
char length;
char data[length];
int crc<watch=length;data, update=WatchCrc32>;
```

pfp.native.watchers.watch_length(*args, **kwargs)

WatchLength - Watch the total length of each of the params.

**Example:** The code below uses the WatchLength update function to update the length field to the length
of the data field
Packer Metadata

Packer metadata allows data structures to be nested inside of transformed/encoded/compressed data. The most common example of this would be gzip-compressed data, that when decompressed also has a defined structure.

Packer metadata can be set in two different ways. In both ways, a packtype key must be set that specifies the structure type that should be used to parse the packed data.

The packing and unpacking function(s) have two ways to be defined:

1. A single function (packer key) that takes an additional parameter that says whether to pack or unpack the data.

2. Two functions that define separate pack and unpack functions. The pack function is optional if you never intend to rebuild the dom.

After packed data has been parsed, the packed data can be accessed via the _ field name::

```c
dom = pfp.parse(...)  
dom.packed_data._.unpacked_field  
...
```

packtype

The packtype key should point to a data type that will be used to parse the packed data. E.g.::

```c
typedef struct {
    int a;
    int b;
} packedData;

struct {
    uchar data[4]<packtype=packedData, ...>;
} main;
```

packer

The packer key should reference a function that can handle both packing and unpacking. The function (native or interpreted) must have the signature::

```c
char[] packerFunction(pack, char data[]) {  
    ...  
    // must return an array of unpacked data
}
```

Note that interpreted packer functions have not been thoroughly tested. Native packers work just fine (see the PackerGZip packer for an example).
**pack**

The `pack` key should be a function that accepts an array of the unpacked data, and returns an array that represents the packed data.

**unpack**

The `unpack` key should be a function that accepts an array of packed data, and returns an array that represents the unpacked data.

### Built-in Pack Functions

```python
pfp.native.packers.pack_gzip(*args, **kwargs)
```

PackGZip - Concat the build output of all params and gzips the resulting data, returning a char array.

Example:
```
char data[0x100]<pack=PackGZip, ...>;
```

```python
pfp.native.packers.packer_gzip(*args, **kwargs)
```

PackerGZip - implements both unpacking and packing. Can be used as the packer for a field. When packing, concats the build output of all params and gzip-compresses the result. When unpacking, concats the build output of all params and gzip-decompresses the result.

Example:
```
The code below specifies that the data field is gzipped and that once decompressed, should be parsed with PACK_TYPE. When building the PACK_TYPE structure, data will be updated with the compressed data.:  
char data[0x100]<packer=PackerGZip, packtype=PACK_TYPE>;
```

- **Pack**  True if the data should be packed, false if it should be unpacked
- **Data**  The data to operate on
- **Returns**  An array

```python
pfp.native.packers.unpack_gzip(*args, **kwargs)
```

UnpackGZip - Concat the build output of all params and gunzips the resulting data, returning a char array.

Example:
```
char data[0x100]<pack=UnpackGZip, ...>;
```

### 2.3 Fields

#### 2.3.1 General

Every declared variable in 010 templates creates a `pfp.fields.Field` instance in memory.
**Naming Convention**

Some may find it annoying having the prefix `_pfp__` affixed to field methods and variables, but I found it more annoying having to access all child fields of a struct via square brackets. The prefix is simply to prevent name collisions so that `__getattr__` can be used to access child fields with dot-notation.

**Parsed Offset**

Parsed offsets of fields are set during object parsing and are re-set each time the main `pfp.fields.Dom` instance is built. This means that operations that should modify the offsets of fields will cause invalid offsets to exist until the main dom is built again.

**Printing**

Use the `pfp.fields.Field._pfp__show` method to return a pretty-printed representation of the field.

**Full Field Paths**

Use the `pfp.fields.Field._pfp__path` method to fetch the full path of the field. E.g. in the template below, the `inner` field would have a full path of `root.nested1.nested2.inner`, and the second element of the `array` field would have a full path of `root.nested1.nested2.array[1]`:

```python
struct {
    struct {
        char inner;
        char array[4];
    } nested2;
    int some_int;
} nested1;
int some_int2;
} root;
```

### 2.3.2 Structs

Structs are the main containers used to add fields to. A `pfp.fields.Dom` instance is the struct that all fields are added to.

### 2.3.3 Field Reference Documentation

```python
class pfp.fields.Field(stream=None, metadata_processor=None)
```
Core class for all fields used in the Pfp DOM.

All methods use the `_pfp__XXX` naming convention to avoid conflicting names used in templates, since struct fields will implement `__getattr__` and `__setattr__` to directly access child fields

```python
_pfp__build(output_stream=None, save_offset=False)
```
Pack this field into a string. If `output_stream` is specified, write the output into the output stream

- **Output_stream** Optional output stream to write the results to
- **Save_offset** If true, the current offset into the stream will be saved in the field
Returns Resulting string if output_stream is not specified. Else the number of bytes written.

_pfp__name = None
The name of the Field

_pfp__parent = None
The parent of the field

_pfp__parse (stream, save_offset=False)
Parse this field from the stream

Stream An IO stream that can be read from
Save_offset Save the offset into the stream

Returns None

_pfp__path ()
Return the full pathname of this field. E.g. given the template below, the a field would have a full path of root.nested.a

```
struct {
    struct {
        char a;
    } nested;
} root;
```

_pfp__set_value (new_val)
Set the new value if type checking is passes, potentially (TODO? reevaluate this) casting the value to something else

New_val The new value

Returns TODO

_pfp__show (level=0, include_offset=False)
Return a representation of this field

Parameters

• level (int) – The indent level of the output
• include_offset (bool) – Include the parsed offsets of this field

_pfp__watch_fields = []
All fields that this field is watching

_pfp__watchers = []
All fields that are watching this field

_pfp__width ()
Return the width of the field (sizeof)

class pfp.fields.Array (width, field_cls, stream=None, metadata_processor=None)
The array field

field_cls = None
The class for items in the array

raw_data = None
The raw data of the array. Note that this will only be set if the array’s items are a core type (E.g. Int, Char, etc)

2.3. Fields
width = -1
    The number of items of the array. len(array_field) also works

class pfp.fields.Struct (stream=None, metadata_processor=None)
The struct field

    _pfp__add_child (name, child, stream=None, overwrite=False)
    Add a child to the Struct field. If multiple consecutive fields are added with the same name, an implicit
    array will be created to store all fields of that name.

    Parameters
    • name (str) – The name of the child
    • child (pfp.fields.Field) – The field to add
    • overwrite (bool) – Overwrite existing fields (False)
    • stream (pfp.bitwrap.BitwrappedStream) – unused, but her for compatibility
      with Union._pfp__add_child

    Returns The resulting field added

    _pfp__children = []
    All children of the struct, in order added

class pfp.fields.Array (width, field_cls, stream=None, metadata_processor=None)
The array field

    field_cls = None
    The class for items in the array

    implicit = False
    If the array is an implicit array or not

    raw_data = None
    The raw data of the array. Note that this will only be set if the array’s items are a core type (E.g. Int, Char,
    etc)

    width = -1
    The number of items of the array. len(array_field) also works

class pfp.fields.BitfieldRW (interp, cls)
Handles reading and writing the total bits for the bitfield data type from the input stream, and correctly applying
endiian and bit direction settings.

    read_bits (stream, num_bits, padded, left_right, endian)
    Return num_bits bits, taking into account endianness and left-right bit directions

    reserve_bits (num_bits, stream)
    Used to “reserve” num_bits amount of bits in order to keep track of consecutive bitfields (or are the
called bitfield groups?).

    E.g.

    struct {
        char a:8, b:8;
        char c:4, d:4, e:8;
    }

    Parameters
    • num_bits (int) – The number of bits to claim
• **stream** ([pfp.bitwrap.BitwrappedStream]) – The stream to reserve bits on

**Returns** If room existed for the reservation

```python
write_bits (stream, raw_bits, padded, left_right, endian)
```
Write the bits. Once the size of the written bits is equal to the number of the reserved bits, flush it to the stream

```python
class pfp.fields.Char(stream=None, bitsize=None, metadata_processor=None, bitfield_rw=None, bitfield_padded=False, bitfield_left_right=False)
```
A field representing a signed char

```python
class pfp.fields.Dom(*args, **kwargs)
```
The main container struct for a template

```python
class pfp.fields.Double(stream=None, bitsize=None, metadata_processor=None, bitfield_rw=None, bitfield_padded=False, bitfield_left_right=False)
```
A field representing a double

```python
class pfp.fields.Enum(stream=None, enum_cls=None, enum_vals=None, bitsize=None, metadata_processor=None, bitfield_rw=None, bitfield_padded=False, bitfield_left_right=False)
```
The enum field class

```python
class pfp.fields.Field(stream=None, metadata_processor=None)
```
The core class for all fields used in the Pfp DOM.
All methods use the _pfp__XXX naming convention to avoid conflicting names used in templates, since struct fields will implement __getattr__ and __setattr__ to directly access child fields

```python
class pfp.fields.Float(stream=None, bitsize=None, metadata_processor=None, bitfield_rw=None, bitfield_padded=False, bitfield_left_right=False)
```
A field representing a float

```python
class pfp.fields.ImplicitArrayWrapper (last_field, implicit_array)
```

```python
class pfp.fields.Int(stream=None, bitsize=None, metadata_processor=None, bitfield_rw=None, bitfield_padded=False, bitfield_left_right=False)
```
A field representing a signed int

```python
class pfp.fields.Int64(stream=None, bitsize=None, metadata_processor=None, bitfield_rw=None, bitfield_padded=False, bitfield_left_right=False)
```
A field representing a signed int64

```python
class pfp.fields.IntBase(stream=None, bitsize=None, metadata_processor=None, bitfield_rw=None, bitfield_padded=False, bitfield_left_right=False)
```
The base class for all integers

```python
class pfp.fields.NumberBase(stream=None, bitsize=None, metadata_processor=None, bitfield_rw=None, bitfield_padded=False, bitfield_left_right=False)
```
The base field for all numeric fields

```python
class pfp.fields.Short(stream=None, bitsize=None, metadata_processor=None, bitfield_rw=None, bitfield_padded=False, bitfield_left_right=False)
```
A field representing a signed short

```python
class pfp.fields.String(stream=None, metadata_processor=None)
```
A null-terminated string. String fields should be interchangeable with char arrays

```python
class pfp.fields.Struct(stream=None, metadata_processor=None)
```
The struct field

## 2.3. Fields

15
2.4 Fuzzing

With the addition of the `pfp.fuzz` module, pfp now supports fuzzing out-of-the-box! (w00t!).

2.4.1 `pfp.fuzz.mutate()` function

pfp contains a `pfp.fuzz.mutate` function that will mutate a provided field. The provided field will most likely just be the resulting dom from calling `pfp.parse`.

The `pfp.fuzz.mutate` function accepts several arguments:

- **field** - The field to fuzz. This does not have to be a `pfp.fields.Dom` object, although in the normal use case it will be.
- **strat_name_or_cls** - The name (or direct class) of the `StratGroup` to use
- **num** - The number of iterations to perform. Defaults to 100
- **at_once** - The number of fields to fuzz at once. Defaults to 1
- **yield_changed** - If true, the mutate generator will yield a tuple of `(mutated_dom, changed_fields)`, where `changed_fields` is a set (not a list) of the fields that were changed. Also note that the yielded set of changed fields can be modified and is no longer needed by the mutate function. Defaults to False
2.4.2 Strategies

My (d0c_s4vage’s) most successful fuzzing approaches have been ones that allowed me to pre-define various fuzzing strategies. This allows one to reuse, tweak existing, or create new strategies specific to each target or attack surface.

StratGroup

pfp strategy groups are containers for sets of field-specific fuzzing strategies. *StratGroups* must define a *unique name*. Strategy groups may also define a custom *filter_fields* method.

E.g. To define a strategy that *only* fuzzes integers, one could do something like this:

```python
class IntegersOnly(pfp.fuzz.StratGroup):
    name = "ints_only"

class IntStrat(pfp.fuzz.FieldStrat):
    klass = pfp.fields.IntBase
    choices = [0, 1, 2, 3]

def filter_fields(self, fields):
    return filter(lambda x: isinstance(x, pfp.fields.IntBase), fields)
```

Then, after parsing some data using a template, the returned *Dom* instance could be mutated like so:

```python
dom = pfp.parse(....)
for mutation in pfp.fuzz.mutate(dom, "ints_only", num=100, at_once=3):
    mutated = mutation._pfp__build()
    # do something with it
```

Note that the string *ints_only* was used as the *strat_name_or_cls* field. We could have also simply passed in the IntegersOnly class:

```python
dom = pfp.parse(....)
for mutation in pfp.fuzz.mutate(dom, IntegersOnly, num=100, at_once=3):
    mutated = mutation._pfp__build()
    # do something with it
```

FieldStrat

*FieldStrats* define a specific fuzzing strategy for a specific field (or set of fields).

All *FieldStrats* must have either a *choices* field defined or a *prob* field defined.

Alternately, the *next_val* function may also be overridden if something more specific is needed.

2.4.3 Fuzzing Reference Documentation

This module contains the base classes used when defining mutation strategies for pfp

```python
class pfp.fuzz.Changer(*orig_data)

    build()
        Apply all changesets to the original data
```

2.4. Fuzzing
change(**kwdsl)
   Intended to be used with a with block. Takes care of pushing and popping the changes, yields the modified data.

pop_changes()
   Return a version of the original data after popping the latest

push_changes(field_set)
   Push a new changeset onto the changeset stack for the provided set of fields.

pfp.fuzz.changeset_mutate(field, strat_name_or_cls, num=100, at_once=1, yield_changed=False, fields_to_modify=None, base_data=None)
   Mutate the provided field (probably a Dom or struct instance) using the strategy specified with strat_name_or_class, yielding num mutations that affect up to at_once fields at once.

   This function will yield back the field after each mutation, optionally also yielding a set of fields that were mutated in that iteration (if yield_changed is True). It should also be noted that the yielded set of changed fields can be modified and is no longer needed by the mutate() function.

   Parameters

   * field (pfp.fields.Field) – The field to mutate (can be anything, not just Dom/Structs)
   * strat_name_or_class – Can be the name of a strategy, or the actual strategy class (not an instance)
   * num (int) – The number of mutations to yield
   * at_once (int) – The number of fields to mutate at once
   * yield_changed (bool) – Yield a list of fields changed along with the mutated dom
   * use_changesets (bool) – If a performance optimization should be used that builds the full output once, and then replaced only the changed fields, including watchers, etc.

   NOTE this does not yet work fully with packed structures (https://pfp.readthedocs.io/en/latest/metadata.html#packer-metadata)

   Returns generator

pfp.fuzz.mutate(field, strat_name_or_cls, num=100, at_once=1, yield_changed=False)
   Mutate the provided field (probably a Dom or struct instance) using the strategy specified with strat_name_or_class, yielding num mutations that affect up to at_once fields at once. This function will yield back the field after each mutation, optionally also yielding a set of fields that were mutated in that iteration (if yield_changed is True). It should also be noted that the yielded set of changed fields can be modified and is no longer needed by the mutate() function.

   :param pfp.fields.Field field: The field to mutate (can be anything, not just Dom/Structs)
   :param strat_name_or_class: Can be the name of a strategy, or the actual strategy class (not an instance)
   :param int num: The number of mutations to yield
   :param int at_once: The number of fields to mutate at once
   :param bool yield_changed: Yield a list of fields changed along with the mutated dom
   :returns: generator

This module contains the base classes used when defining fuzzing strategies for pfp

class pfp.fuzz.strats.FieldStrat
   A FieldStrat is used to define a fuzzing strategy for a specific field (or list of fields). A list of choices can be defined, or a set or probabilities that will yield

   choices = None
      An enumerable of new value choices to choose from when mutating.

   This can also be a function/callable that returns an enumerable of choices. If it is a callable, the currently-being-fuzzed field will be passed in as a parameter.
**klass = None**
The class this strategy should be applied to. Can be a `pfp.fields.field` class (or subclass) or a string of the class name.

Note that strings for the class name will only apply to direct instances of that class and not instances of subclasses.

Can also be a list of classes or class names.

**mutate (field)**
Mutarate the given field, modifying it directly. This is not intended to preserve the value of the field.

  **Field** The `pfp.fields.Field` instance that will receive the new value

**next_val (field)**
Return a new value to mutate a field with. Do not modify the field directly in this function. Override the `mutate()` function if that is needed (the field is only passed into this function as a reference).

  **Field** The `pfp.fields.Field` instance that will receive the new value. Passed in for reference only.

  **Returns** The next value for the field

**prob = None**
An enumerable of probabilities used to choose from when mutating E.g.:

```python
[  
  (0.50, 0xffff), # 50% of the time it should be the value 0xffff
  (0.25, xrange(0, 0x100)), # 25% of the time it should be in the range [0, 0x100)  
  (0.20, [0, 0xff, 0x100]), # 20% of the time it should be on of 0, 0xff, or 0x100  
  (0.05, {"min": 0, "max": 0x1000}), # 5% of the time, generate a number in [min, max)  
]
```

NOTE that the percentages need to add up to 100.

This can also be a function/callable that returns an probabilities list. If it is a callable, the currently-being-fuzzed field will be passed in as a parameter.

**exception pfp.fuzz.strats.MutationError**

**pfp.fuzz.strats.STRATS = {None: <class 'pfp.fuzz.strats.StratGroup'>, 'basic': <class 'pfp.fuzz.basic.BasicStrat'>}**
Stores information on registered StatGroups

**class pfp.fuzz.strats.StratGroup**
StatGroups choose which sub-fields should be mutated, and which FieldStrat should be used to do the mutating.

The `filter_fields` method is intended to be overridden to provide custom filtering of child leaf fields should be mutated.

**filter_fields (field_list)**
Intended to be overridden. Should return a list of fields to be mutated.

  **Field_list** The list of fields to filter

**get_field_strat (field)**
Return the strategy defined for the field.

  **Field** The field

  **Returns** The FieldStrat for the field or None

---

2.4. Fuzzing 19
The unique name of the fuzzing strategy group. Can be used as the `strat_name_or_cls` parameter to the `pfp.fuzz.mutate()` function.

Return a list of leaf fields that should be mutated. If the field passed in is a leaf field, it will be returned in a list.

A metaclass for StratGroups that tracks subclasses of the StatGroup class.

Return the strategy identified by its name. If `name_or_class` is a class, it will be simply returned.

This module defines basic mutation strategies.

A basic strategy that has FieldStrats (field strategies) defined for every field type. Nothing fancy, just basic.

class Double

    klass
    alias of `pfp.fields.Double`

class Enum

    klass
    alias of `pfp.fields.Enum`

class Float

    klass
    alias of `pfp.fields.Float`

class Int

class String

    klass
    alias of `pfp.fields.String`

    next_val (field)
    Return a new value to mutate a field with. Do not modify the field directly in this function. Override the `mutate()` function if that is needed (the field is only passed into this function as a reference).
    
    Field The `pfp.fields.Field` instance that will receive the new value. Passed in for reference only.
    
    Returns The next value for the field

2.5 Debugger

2.5.1 QuickStart

Pfp comes with a built-in debugger. You can drop into the interactive debugger by calling the `Int3()` function within a template.

All commands are documented below in the debug reference documentation. Command methods begin with `do_`. 
2.5.2 Internals

While the pfp interpreter is handling AST nodes, it decides if a node can be “breaked” on using the `_node_is_breakable` method. If the interpreter is in a debug state, and the current node can be breaked on, the user will be dropped into the interactive debugger.

2.5.3 Debugger Reference Documentation

```python
class pfp.dbg.PfpDbg(interp):
    The pfp debugger cmd.Cmd class

    def default(self, line):
        Called on an input line when the command prefix is not recognized.
        If this method is not overridden, it prints an error message and returns.

    def do_EOF(self, args):
        The eof command

    def do_continue(self, args):
        Continue the interpreter

    def do_eval(self, args):
        Eval the user-supplied statement. Note that you can do anything with this command that you can do in a template.
        The resulting value of your statement will be displayed.

    def do_list(self, args):
        List the current location in the template

    def do_next(self, args):
        Step over the next statement

    def do_peek(self, args):
        Peek at the next 16 bytes in the stream:
        Example:
        The peek command will display the next 16 hex bytes in the input stream:
        
        pfp> peek
        89 50 4e 47 0d 0a 1a 0a 00 00 00 0d 49 48 44 52 .PNG.......IHDR

    def do_quit(self, args):
        The quit command

    def do_s(self, args):
        Step into the next statement

    def do_show(self, args):
        Show the current structure of __root (no args), or show the result of the expression (something that can be eval’d).

    def do_step(self, args):
        Step INTO the next statement

    def do_x(self, args):
        Show the current structure of __root (no args), or show the result of the expression (something that can be eval’d).
```

2.5. Debugger
postcmd \texttt{(stop, line)}

Hook method executed just after a command dispatch is finished.

preloop()

Hook method executed once when the cmdloop() method is called.

\texttt{pfp.native.dbg.\texttt{int3}(*args, **kwargs)}

Define the \texttt{int3()} function in the interpreter. Calling \texttt{Int3()} will drop the user into an interactive debugger.

### 2.6 Interpreter

The Pfp interpreter is quite simple: it uses \texttt{py010parser} to parse the template into an abstract-syntax-tree, and then handles each of the nodes in the tree appropriately.

The main method for handling nodes is the \texttt{_handle_node} function. The \texttt{_handle_node} function performs basic housekeeping, logging, decides if the user should be dropped into the interactive debugger, and of course, handles the node itself.

If a methods are not implemented to handle a certain AST node, an \texttt{pfp.errors.UnsupportedASTNode} error will be raised. Implemented methods to handle AST node types are found in the \texttt{_node_switch} dict:

```python
self._node_switch = {
    AST.FileAST: self._handle_file_ast,
    AST.Decl: self._handle_decl,
    AST.TypeDecl: self._handle_type_decl,
    AST.ByRefDecl: self._handle_byref_decl,
    AST.Struct: self._handle_struct,
    AST.Union: self._handle_union,
    AST.StructRef: self._handle_struct_ref,
    AST.IdentifierType: self._handle_identifier_type,
    AST.Typedef: self._handle_typedef,
    AST.Constant: self._handle_constant,
    AST.BinaryOp: self._handle_binary_op,
    AST.Assignment: self._handle_assignment,
    AST.ID: self._handle_id,
    AST.UnaryOp: self._handle_unary_op,
    AST.FuncDef: self._handle_func_def,
    AST.FuncCall: self._handle_func_call,
    AST.FuncDecl: self._handle_func_decl,
    AST.ParamList: self._handle_param_list,
    AST.ExprList: self._handle_expr_list,
    AST.Compound: self._handle_compound,
    AST.Return: self._handle_return,
    AST.ArrayDecl: self._handle_array_decl,
    AST.InitList: self._handle_init_list,
    AST.If: self._handle_if,
    AST.For: self._handle_for,
    AST.While: self._handle_while,
    AST.DeclList: self._handle_decl_list,
    AST.Break: self._handle_break,
    AST.Continue: self._handle_continue,
    AST.ArrayRef: self._handle_array_ref,
    AST.Enum: self._handleEnum,
    AST.EmptyStatement: self._handle_empty_statement,
}
```

(continues on next page)
2.6.1 Interpreter Reference Documentation

Python format parser

```
pfp.interp.LazyField(lookup_name, scope)
```

Super non-standard stuff here. Dynamically changing the base class using the scope and the lazy name when the class is instantiated. This works as long as the original base class is not directly inheriting from object (which we’re not, since our original base class is fields.Field).

```
class pfp.interp.PfpInterp(debug=False, parser=None, int3=True)
```

```
classmethod add_native(name, func, ret, interp=None, send_interp=False)
```

Add the native python function `func` into the pfp interpreter with the name `name` and return value `ret` so that it can be called from within a template script.

**Note:** The `@native` decorator exists to simplify this.

All native functions must have the signature `def func(params, ctxt, scope, stream, coord [,interp])`, optionally allowing an interpreter param if `send_interp` is True.

**Example:**

The example below defines a function `Sum` using the `add_native` method.

```
import pfp.fields
from pfp.fields import PYVAL

def native_sum(params, ctxt, scope, stream, coord):
    return PYVAL(params[0]) + PYVAL(params[1])

pfp.interp.PfpInterp.add_native("Sum", native_sum, pfp.fields.Int64)
```

**Parameters**

- `name (basestring)` – The name the function will be exposed as in the interpreter.
- `func (function)` – The native python function that will be referenced.
- `ret (type(pfp.fields.Field))` – The field class that the return value should be cast to.
- `interp (pfp.interp.PfpInterp)` – The specific pfp interpreter the function should be defined in.
- `send_interp (bool)` – If true, the current pfp interpreter will be added as an argument to the function.

```
classmethod add_predefine(template)
```

Add a template that should be run prior to running any other templates. This is useful for predefining types, etc.
Parameters `template` *(basestring)* – The template text (unicode is also fine here)

```python
cont()
```
Continue the interpreter

```python
classmethod define_natives()
```
Define the native functions for PFP

```python
eval(*statement*, *ctxt=None*)
```
Eval a single statement (something returnable)

```python
get_bitfield_direction()
```
Return if the bitfield direction

**Note:** This should be applied AFTER taking into account endianness.

```python
get_bitfield_padded()
```
Return if the bitfield input/output stream should be padded

**Returns** True/False

```python
get_curr_lines()
```
Return the current line number in the template, as well as the surrounding source lines

```python
get_filename()
```
Return the filename of the data that is currently being parsed

**Returns** The name of the data file being parsed.

```python
get_types()
```
Return a types object that will contain all of the typedef structs’ classes.

**Returns** Types object

**Example:**

Create a new PNG_CHUNK object from a PNG_CHUNK type that was defined in a template:

```python
types = interp.get_types() chunk = types.PNG_CHUNK()
```

```python
load_template(*template*)
```
Load a template and all required predefines into this interpreter. Future calls to `parse` will not require the template to be parsed.

```python
parse(*stream*, *template=None*, *predefines=True*, *orig_filename=None*, *keep_successful=False*, *printf=True*)
```
Parse the data stream using the template (e.g. parse the 010 template and interpret the template using the stream as the data source).

**Stream** The input data stream

**Template** The template to parse the stream with

**Keep_successful** Return whatever was successfully parsed before an error. `_pfp__error` will contain the exception (if one was raised)

**Parameters** `printf` *(bool)* – If `False`, printfs will be noops (default=‘True‘)

**Returns** Pfp Dom

```python
set_bitfield_direction(*val*)
```
Set the bitfields to parse from left to right (1), the default (None), or right to left (-1)
set_bitfield_padded(val)
    Set if the bitfield input/output stream should be padded
    
    Val True/False
    Returns None

set_break(break_type)
    Set if the interpreter should break.
    
    Returns TODO

step_into()
    Step over/into the next statement

step_over()
    Perform one step of the interpreter

class pfp.interp.PfpTypes(interp, scope)
    A class to hold all typedefd types in a template. Note that types are instantiated by having them parse a null-stream. This means that type creation will not work correctly for complicated structs that have internal control-flow

class pfp.interp.Scope(logger, parent=None)
    A class to keep track of the current scope of the interpreter

add_local(field_name, field)
    Add a local variable in the current scope
        
        Field_name The field's name
        Field The field
        Returns None

add_refd_struct_or_union(name, refd_name, interp, node)
    Add a lazily-looked up typedefd struct or union
        
        Name name of the typedefd struct/union
        Node the typedef node
        Interp the 010 interpreter

add_type(new_name, orig_names)
    Record the typedefd name for orig_names. Resolve orig_names to their core names and save those.
        
        New_name TODO
        Orig_names TODO
        Returns TODO

add_type_class(name, cls)
    Store the class with the name

add_type_struct_or_union(name, interp, node)
    Store the node with the name. When it is instantiated, the node itself will be handled.
        
        Name name of the typedefd struct/union
        Node the union/struct node
        Interp the 010 interpreter

add_var(field_name, field, root=False)
    Add a var to the current scope (vars are fields that parse the input stream)
Field_name  TODO
Field  TODO
Returns  TODO
clear_meta ()
Clear metadata about the current statement
clon()  
Return a new Scope object that has the curr_scope pinned at the current one :returns: A new scope object
get_id (name, recurse=True)
Get the first id matching name. Will either be a local or a var.
   Name  TODO
   Returns  TODO
get_local (name, recurse=True)
Get the local field (search for it) from the scope stack. An alias for get_var
   Name  The name of the local field
get_meta (meta_name)
Get the current meta value named meta_name
get_type (name, recurse=True)
Get the names for the typename (created by typedef)
   Name  The typedef’d name to resolve
   Returns  An array of resolved names associated with the typedef’d name
get_var (name, recurse=True)
Return the first var of name name in the current scope stack (remember, vars are the ones that parse the input stream)
   Name  The name of the id
   Recurse  Whether parent scopes should also be searched (defaults to True)
   Returns  TODO
level ()
Return the current scope level
pop ()
Leave the current scope :returns: TODO
pop_meta (name)
Pop metadata about the current statement from the metadata stack for the current statement.
   Name  The name of the metadata
push (new_scope=None)
Create a new scope :returns: TODO
push_meta (meta_name, meta_value)
Push metadata about the current statement onto the metadata stack for the current statement. Mostly used for tracking integer promotion and casting types
pfp.interp.StructUnionTypeRef (curr_scope, typedef_name, refd_name, interp, node)
Create a typedef that resolves itself dynamically. This is needed in situations like:
```c
struct MY_STRUCT {
    char magic[4];
    unsigned int filesize;
};
typedef struct MY_STRUCT ME;
LittleEndian();
ME s;
```

The typedef ME is handled before the MY_STRUCT declaration actually occurs. The typedef value for ME should not the empty struct that is resolved, but should be a dynamically-looked up struct definition when a ME instance is actually declared.

Python format parser

```python
pfp.interp.LazyField(lookup_name, scope)
```

Super non-standard stuff here. Dynamically changing the base class using the scope and the lazy name when the class is instantiated. This works as long as the original base class is not directly inheriting from object (which we’re not, since our original base class is fields.Field).

```python
class pfp.interp.PfpInterp(debug=False, parser=None, int3=True)
```

```
classmethod add_native(name, func, ret, interp=None, send_interp=False)
```

Add the native python function func into the pfp interpreter with the name name and return value ret so that it can be called from within a template script.

**Note:** The @native decorator exists to simplify this.

All native functions must have the signature `def func(params, ctxt, scope, stream, coord [,interp]),` optionally allowing an interpreter param if send_interp is True.

Example:

The example below defines a function Sum using the add_native method.

```python
import pfp.fields
from pfp.fields import PYVAL

def native_sum(params, ctxt, scope, stream, coord):
    return PYVAL(params[0]) + PYVAL(params[1])
pfp.interp.PfpInterp.add_native("Sum", native_sum, pfp.fields.Int64)
```

**Parameters**

- **name** (*basestring*) – The name the function will be exposed as in the interpreter.
- **func** (*function*) – The native python function that will be referenced.
- **ret** (*type (pfp.fields.Field)*) – The field class that the return value should be cast to.
- **interp** (*pfp.interp.PfpInterp*) – The specific pfp interpreter the function should be defined in.
- **send_interp** (*bool*) – If true, the current pfp interpreter will be added as an argument to the function.
**classmethod add_predefine**(template)

Add a template that should be run prior to running any other templates. This is useful for predefining types, etc.

**Parameters** template (basestring) – The template text (unicode is also fine here)

**cont()**

Continue the interpreter

**classmethod define_natives()**

Define the native functions for PFP

**eval**(statement, ctxt=None)

Eval a single statement (something returnable)

**get_bitfield_direction()**

Return if the bitfield direction

---

**Note:** This should be applied AFTER taking into account endianness.

**get_bitfield_padded()**

Return if the bitfield input/output stream should be padded

**Returns** True/False

**get_curr_lines()**

Return the current line number in the template, as well as the surrounding source lines

**get_filename()**

Return the filename of the data that is currently being parsed

**Returns** The name of the data file being parsed.

**get_types()**

Return a types object that will contain all of the typedef structs’ classes.

**Returns** Types object

**Example:**

Create a new PNG_CHUNK object from a PNG_CHUNK type that was defined in a template:

types = interp.get_types() chunk = types.PNG_CHUNK()

**load_template**(template)

Load a template and all required predefines into this interpreter. Future calls to **parse** will not require the template to be parsed.

**parse**(stream, template=None, predefines=True, orig_filename=None, keep_successful=False, printf=True)

Parse the data stream using the template (e.g. parse the 010 template and interpret the template using the stream as the data source).

**Stream** The input data stream

**Template** The template to parse the stream with

**Keep_successful** Return whatever was successfully parsed before an error. _pfp__error will contain the exception (if one was raised)

**Parameters** printf (bool) – If False, printfs will be noops (default=’True’)

**Returns** Pfp Dom
**set_bitfield_direction** *(val)*
Set the bitfields to parses from left to right (1), the default (None), or right to left (-1)

**set_bitfield_padded** *(val)*
Set if the bitfield input/output stream should be padded

- **Val** True/False
- **Returns** None

**set_break** *(break_type)*
Set if the interpreter should break.

- **Returns** TODO

**step_into** ()
Step over/into the next statement

**step_over** ()
Perform one step of the interpreter

```python
class pfp.interp.PfpTypes(interp, scope)
A class to hold all typedefd types in a template. Note that types are instantiated by
having them parse a null-stream. This means that type creation will not work correctly for
complicated structs that have internal control-flow
```

```python
class pfp.interp.Scope(logger, parent=None)
A class to keep track of the current scope of the interpreter
```

**add_local** *(field_name, field)*
Add a local variable in the current scope

- **Field_name** The field’s name
- **Field** The field
- **Returns** None

**add_refd_struct_or_union** *(name, refd_name, interp, node)*
Add a lazily-looked up typedef struct or union

- **Name** name of the typedefd struct/union
- **Node** the typedefd node
- **Interp** the 010 interpreter

**add_type** *(new_name, orig_names)*
Record the typedefd name for orig_names. Resolve orig_names to their core names and save those.

- **New_name** TODO
- **Orig_names** TODO
- **Returns** TODO

**add_type_class** *(name, cls)*
Store the class with the name

**add_type_struct_or_union** *(name, interp, node)*
Store the node with the name. When it is instantiated, the node itself will be handled.

- **Name** name of the typedefd struct/union
- **Node** the union/struct node
- **Interp** the 010 interpreter
**add_var**(*field_name, field, root=False*)

Add a var to the current scope (vars are fields that parse the input stream)

- **Field_name** TODO
- **Field** TODO
- **Returns** TODO

**clear_meta**()

Clear metadata about the current statement

**clone**()

Return a new Scope object that has the curr_scope pinned at the current one :returns: A new scope object

**get_id**(*name, recurse=True*)

Get the first id matching *name*. Will either be a local or a var.

- **Name** TODO
- **Returns** TODO

**get_local**(*name, recurse=True*)

Get the local field (search for it) from the scope stack. An alias for **get_var**

- **Name** The name of the local field

**get_meta**(*meta_name*)

Get the current meta value named *meta_name*

**get_type**(*name, recurse=True*)

Get the names for the typename (created by typedef)

- **Name** The typedef’d name to resolve
- **Returns** An array of resolved names associated with the typedef’d name

**get_var**(*name, recurse=True*)

Return the first var of name *name* in the current scope stack (remember, vars are the ones that parse the input stream)

- **Name** The name of the id
- **Recurse** Whether parent scopes should also be searched (defaults to True)
- **Returns** TODO

**level**()

Return the current scope level

**pop**()

Leave the current scope :returns: TODO

**pop_meta**(*name*)

Pop metadata about the current statement from the metadata stack for the current statement.

- **Name** The name of the metadata

**push**(*new_scope=None*)

Create a new scope :returns: TODO

**push_meta**(*meta_name, meta_value*)

Push metadata about the current statement onto the metadata stack for the current statement. Mostly used for tracking integer promotion and casting types
pfp.interp.StructUnionTypeRef (curr_scope, typedef_name, refd_name, interp, node)

Create a typedef that resolves itself dynamically. This is needed in situations like:

```
struct MY_STRUCT {
    char magic[4];
    unsigned int filesize;
};
typedef struct MY_STRUCT ME;
LittleEndian();
ME s;
```

The typedef ME is handled before the MY_STRUCT declaration actually occurs. The typedef value for ME should not the empty struct that is resolved, but should be a dynamically-looked up struct definition when a ME instance is actually declared.

### 2.7 Functions

Functions in pfp can either be defined natively in python, or in the template script itself.

#### 2.7.1 Native Functions

Two main methods exist to add native python functions to the pfp interpreter:

1. The `@native` decorator
2. The `add_native` method

Follow the links above for detailed information.

#### 2.7.2 Interpreted Functions

Interpreted functions can declared as you normally would in an 010 template (basically c-style syntax).

Functions are hoisted to the top of the scope they are declared in. E.g. the following script is valid:

```
HelloWorld(10);

typedef unsigned short custom_short;
void HelloWorld(custom_short arg1) {
    Printf("Hello World, \%d", arg1);
}
```

#### 2.7.3 Functions Reference Documentation

```python
class pfp.functions.Function (return_type, params, scope)
    A class to maintain function state and arguments

class pfp.functions.NativeFunction (name, func, ret, send_interp=False)
    A class for native functions

class pfp.functions.ParamClsWrapper (param_cls)
    This is a temporary wrapper around a param class that can store temporary information, such as byref values
```
class pfp.functions.ParamList(params)
    Used for when a function is actually called. See ParamListDef for how function definitions store function parameter definitions

class pfp.functions.ParamListDef(params, coords)
    docstring for ParamList

    instantiate(scope, args, interp)
        Create a ParamList instance for actual interpretation

        Args  TODO
        Returns A ParamList object

pfp.native.native(name, ret, interp=None, send_interp=False)
    Used as a decorator to add the decorated function to the pfp interpreter so that it can be used from within scripts.

    Parameters
    • name (str) – The name of the function as it will be exposed in template scripts.
    • ret (pfp.fields.Field) – The return type of the function (a class)
    • interp (pfp.interp.PfpInterp) – The specific interpreter to add the function to
    • send_interp (bool) – If the current interpreter should be passed to the function.

    Examples:

    The example below defines a Sum function that will return the sum of all parameters passed to the function:

    ```python
    from pfp.fields import PYVAL
    @native(name="Sum", ret=pfp.fields.Int64)
    def sum_numbers(params, ctxt, scope, stream, coord):
        res = 0
        for param in params:
            res += PYVAL(param)
        return res
    ```

    The code below is the code for the Int3 function. Notice that it requires that the interpreter be sent as a parameter:

    ```python
    @native(name="Int3", ret=pfp.fields.Void, send_interp=True)
    def int3(params, ctxt, scope, stream, coord, interp):
        if interp._no_debug:
            return
        if interp._int3:
            interp.debugger = PfpDbg(interp)
            interp.debugger.cmdloop()
    ```

2.8 Bitstream

In order to implement the functionality that 010 editor has of treating the entire stream as a bitstream, a stream-wrapping class (pfp.bitwrap.BitwrappedStream) was made to allow a normal stream to tread like a limited bit stream.

This may be useful in other applications outside of pfp.
2.8.1 BitwrappedStream Reference Documentation

class pfp.bitwrap.BitwrappedStream(stream)
    A stream that wraps other streams to provide bit-level access

    close()
        Close the stream

    flush()
        Flush the stream

    is_eof()
        Return if the stream has reached EOF or not without discarding any unflushed bits

        Returns True/False

    isatty()
        Return if the stream is a tty

    read(num)
        Read num number of bytes from the stream. Note that this will automatically resets/ends the current bit-
        reading if it does not end on an even byte AND self.padded is True. If self.padded is True, then
        the entire stream is treated as a bitstream.

        Num number of bytes to read

        Returns the read bytes, or empty string if EOF has been reached

    read_bits(num)
        Read num number of bits from the stream

        Num number of bits to read

        Returns a list of num bits, or an empty list if EOF has been reached

    seek(pos, seek_type=0)
        Seek to the specified position in the stream with seek_type. Unflushed bits will be discarded in the case of
        a seek.

        The stream will also keep track of which bytes have and have not been consumed so that the dom will
        capture all of the bytes in the stream.

        Pos offset

        Seek_type direction

        Returns TODO

    size()
        Return the size of the stream, or -1 if it cannot be determined.

    tell()
        Return the current position in the stream (ignoring bit position)

        Returns int for the position in the stream

    tell_bits()
        Return the number of bits into the stream since the last whole byte.

        Returns int

    unconsumed_ranges()
        Return an IntervalTree of unconsumed ranges, of the format (start, end] with the end value not being
        included
write(data)
  Write data to the stream
        Data the data to write to the stream
  Returns None
write_bits(bits)
  Write the bits to the stream.
        Add the bits to the existing unflushed bits and write complete bytes to the stream.

exception pfp.bitwrap.EOFError
pfp.bitwrap.bits_to_bytes(bits)
  Convert the bit list into bytes. (Assumes bits is a list whose length is a multiple of 8)
pfp.bitwrap.byte_to_bits(b)
  Convert a byte into bits
pfp.bitwrap.bytes_to_bits(bytes_)
  Convert bytes to a list of bits

2.9 Differences Between 010 and pfp

This section documents the known differences between pfp and 010 editor.

2.9.1 Duplicate Arrays

TLDR: Pfp does not [yet] support non-consecutive duplicate arrays. Consecutive duplicate arrays are fully supported. First, some definitions and back story.

Duplicate arrays are what occurs when multiple variables of the same name are declared in the same scope. E.g.:

```c
int x;
int x;
if (x[0] == x[1] || x[0] == x) {
    printf("Same!");
}
```

The 010 template script above declares x twice, creating a duplicate, or as pfp originally called it, an implicit array. Notice the two comparisons - they actually perform the same comparison:

```c
x[0] != x[1]
```

and

```c
x[0] == x
```

In 010, if the duplicate/implicit array is referenced without indexing, the most recently parsed field in the duplicate array is returned. I.e., it’s treated as a normal field and not an array. However, if indexing is done on the duplicate array variable, the variable is treated as an array.

Below is a quote on duplicate arrays from the 010 Editor documentation:

When writing a template, regular arrays can be declaring using the same syntax as scripts (see Arrays and Strings). However, 010 Editor has a syntax that allows arrays to be built in a special way. When declaring template variables, multiple copies of the same variable can be declared. For example:
010 Editor allows you to treat the multiple declarations of the variable as an array (this is called a Duplicate Array). In this example, x[0] could be used to reference the first occurrence of x and x[1] could be used to reference the second occurrence of x. Duplicate arrays can even be defined with for or while loops. For example:

```cpp
local int i;
for(i = 0; i < 5; i++)
    int x;
```

This breaks down in pfp when non-consecutive arrays are created, as is done in the first code sample from the 010 Editor documentation above. Issue #111 tracks the effort to add support for non-consecutive duplicate arrays.

```python
pfp.create_interp(template_file=None, template=None)
Create an Interp instance with the template preloaded

Template template contents (str)
Template_file template file path

Returns Interp
```

```python
pfp.parse(data=None, template=None, data_file=None, template_file=None, interp=None, debug=False, predefines=True, int3=True, keep_successful=False, printf=True)
Parse the data stream using the supplied template. The data stream WILL NOT be automatically closed.

Data Input data, can be either a string or a file-like object (StringIO, file, etc)
Template template contents (str)
Data_file PATH to the data to be used as the input stream
Template_file template file path
Interp the interpreter to be used (a default one will be created if None)
Debug if debug information should be printed while interpreting the template (false)
Predefines if built-in type information should be inserted (true)
Int3 if debugger breaks are allowed while interpreting the template (true)
Keep_successful return any successfully parsed data instead of raising an error. If an error occurred and keep_successful is True, then _pfp__error will be contain the exception object
Printf if False, all calls to Printf will be noops. (default="True")

Returns pfp DOM
```
CHAPTER 3

Indices and tables

- genindex
- modindex
- search
Python Module Index

p
  pfp, 35
  pfp.bitwrap, 33
  pfp.dbg, 21
  pfp.fields, 14
  pfp.functions, 31
  pfp.fuzz, 17
  pfp.fuzz.basic, 20
  pfp.fuzz.strats, 18
  pfp.interp, 27
  pfp.native, 32
  pfp.native.dbg, 22
  pfp.native.packers, 11
  pfp.native.watchers, 9
Symbols

_pfp__add_child () (pfp.fields.Struct method), 14
_pfp__build () (pfp.fields.Field method), 12
_pfp__children (pfp.fields.Struct attribute), 14
_pfp__name (pfp.fields.Field attribute), 13
_pfp__parent (pfp.fields.Field attribute), 13
_pfp__parse () (pfp.fields.Field method), 13
_pfp__set_value () (pfp.fields.Field method), 13
_pfp__watch_fields (pfp.fields.Field attribute), 13
_pfp__watchers (pfp.fields.Field attribute), 13
_pfp__width () (pfp.fields.Field method), 13
add_local () (pfp.interp.Scope method), 25, 29
add_native () (pfp.interp.PfpInterp class method), 23, 27
add_predefine () (pfp.interp.PfpInterp class method), 23, 27
add_refd_struct_or_union () (pfp.interp.Scope method), 25, 29
add_type () (pfp.interp.Scope method), 25, 29
add_type_class () (pfp.interp.Scope method), 25, 29
add_type_struct_or_union () (pfp.interp.Scope method), 25, 29
add_var () (pfp.interp.Scope method), 25, 29
Array (class in pfp.fields), 13, 14
BasicStrat (class in pfp.fuzz.basic), 20
BasicStrat.Double (class in pfp.fuzz.basic), 20
BasicStrat.Enum (class in pfp.fuzz.basic), 20
BasicStrat.Float (class in pfp.fuzz.basic), 20
BasicStrat.Int (class in pfp.fuzz.basic), 20
BasicStrat.String (class in pfp.fuzz.basic), 20
BitfieldRW (class in pfp.fields), 14
bits_to_bytes () (in module pfp.bitwrap), 34
BitwrappedStream (class in pfp.bitwrap), 33
build () (pfp.fuzz.Changer method), 17
byte_to_bits () (in module pfp.bitwrap), 34
bytes_to_bits () (in module pfp.bitwrap), 34
C
cchange () (pfp.fuzz.Changer method), 17
Changer (class in pfp.fuzz), 17
changeset_mutate () (in module pfp.fuzz), 18
Char (class in pfp.fields), 15
choices (pfp.fuzz.strats.FieldStrat attribute), 18
clear_meta () (pfp.interp.Scope method), 26, 30
close () (pfp.bitwrap.BitwrappedStream method), 33
cnt () (pfp.interp.PfpInterp method), 24, 28
create_interp () (in module pfp), 35
D
default () (pfp.dbg.PfpDbg method), 21
define_natives () (pfp.interp.PfpInterp class method), 24, 28
do_continue () (pfp.dbg.PfpDbg method), 21
do_EOF () (pfp.dbg.PfpDbg method), 21
do_eval () (pfp.dbg.PfpDbg method), 21
do_list () (pfp.dbg.PfpDbg method), 21
do_next () (pfp.dbg.PfpDbg method), 21
do_peek () (pfp.dbg.PfpDbg method), 21
do_quit () (pfp.dbg.PfpDbg method), 21
do_s () (pfp.dbg.PfpDbg method), 21
do_show () (pfp.dbg.PfpDbg method), 21
do_step () (pfp.dbg.PfpDbg method), 21
do_x () (pfp.dbg.PfpDbg method), 21
Dom (class in pfp.fields), 15
Double (class in pfp.fields), 15
E
Enum (class in pfp.fields), 15
EOFError, 34
eval () (pfp.interp.PfpInterp method), 24, 28
F
Field (class in pfp.fields), 12, 15
field_cls (pfp.fields.Array attribute), 13, 14
FieldStrat (class in pfp.fuzz.strats), 18
filter_fields () (pfp.fuzz.strats.StratGroup method), 19
Float (class in pfp.fields), 15
flush () (pfp.bitwrap.BitwrappedStream method), 33
Function (class in pfp.functions), 31

G
get_bitfield_direction () (pfp.interp.PfpInterp method), 24, 28
get_bitfield_padded () (pfp.interp.PfpInterp method), 24, 28
get_curr_lines () (pfp.interp.PfpInterp method), 24, 28
get_field_strat () (pfp.fuzz.strats.StratGroup method), 19
get_filename () (pfp.interp.PfpInterp method), 24, 28
get_id () (pfp.interp.Scope method), 26, 30
get_local () (pfp.interp.Scope method), 26, 30
get_meta () (pfp.interp.Scope method), 26, 30
get_strategy () (in module pfp.fuzz.strats), 20
get_type () (pfp.interp.Scope method), 26, 30
get_types () (pfp.interp.PfpInterp method), 24, 28
get_var () (pfp.interp.Scope method), 26, 30

I
implicit (pfp.fields.Array attribute), 14
ImplicitArrayWrapper (class in pfp.fields), 15
instantiate () (pfp.functions.ParamListDef method), 32
Int (class in pfp.fields), 15
Int3 () (in module pfp.native.dbg), 22
Int64 (class in pfp.fields), 15
IntBase (class in pfp.fields), 15
is_eof () (pfp.bitwrap.BitwrappedStream method), 33
isatty () (pfp.bitwrap.BitwrappedStream method), 33

K
klass (pfp.fuzz.basic.BasicStrat.Double attribute), 20
class (pfp.fuzz.basic.BasicStrat.Enum attribute), 20
class (pfp.fuzz.basic.BasicStrat.Float attribute), 20
class (pfp.fuzz.basic.BasicStrat.String attribute), 20
class (pfp.fuzz.strats.FieldStrat attribute), 18

L
LazyField () (in module pfp.interp), 23, 27
level () (pfp.interp.Scope method), 26, 30
load_template () (pfp.interp.PfpInterp method), 24, 28

M
mutate () (in module pfp.fuzz), 18
mutate () (pfp.fuzz.strats.FieldStrat method), 19
MutationError, 19

N
name (pfp.fuzz.strats.StratGroup attribute), 19
native () (in module pfp.native), 32
NativeFunction (class in pfp.functions), 31
next_val () (pfp.fuzz.basic.BasicStrat.String method), 20
next_val () (pfp.fuzz.strats.FieldStrat method), 19
NumberBase (class in pfp.fields), 15

P
pack_gzip () (in module pfp.native.packers), 11
packer_gzip () (in module pfp.native.packers), 11
ParamClsWrapper (class in pfp.functions), 31
ParamList (class in pfp.functions), 31
ParamListDef (class in pfp.functions), 32
parse () (in module pfp), 35
parse () (pfp.interp.PfpInterp method), 24, 28
ppf (module), 35
ppf.bitwrap (module), 33
ppf.dbg (module), 21
ppf.fields (module), 14
ppf.functions (module), 31
ppf.fuzz (module), 17
ppf.fuzz.basic (module), 20
ppf.fuzz.strats (module), 18
ppf.interp (module), 23, 27
ppf.native (module), 32
ppf.native.dbg (module), 22
ppf.native.packers (module), 11
ppf.native.watchers (module), 9
PfpDbg (class in pfp.dbg), 21
PfpInterp (class in pfp.interp), 23, 27
PfpTypes (class in pfp.interp), 25, 29
pop () (pfp.interp.Scope method), 26, 30
pop_changes () (pfp.fuzz.Changer method), 18
pop_meta () (pfp.interp.Scope method), 26, 30
postcmd () (pfp.dbg.PfpDbg method), 21
preloop () (pfp.dbg.PfpDbg method), 22
prob (pfp.fuzz.strats.FieldStrat attribute), 19
push () (pfp.interp.Scope method), 26, 30
push_changes () (pfp.fuzz.Changer method), 18
push_meta () (pfp.interp.Scope method), 26, 30

R
raw_data (pfp.fields.Array attribute), 13, 14
read () (pfp.bitwrap.BitwrappedStream method), 33
read_bits () (pfp.bitwrap.BitwrappedStream method), 33
read_bits() (pfp.fields.BitfieldRW method), 14
reserve_bits() (pfp.fields.BitfieldRW method), 14

S
Scope (class in pfp.interp), 25, 29
seek() (pfp.bitwrap.BitwrappedStream method), 33
set_bitfield_direction() (pfp.interp.PfpInterp method), 24, 28
set_bitfield_padded() (pfp.interp.PfpInterp method), 24, 29
set_break() (pfp.interp.PfpInterp method), 25, 29
Short (class in pfp.fields), 15
size() (pfp.bitwrap.BitwrappedStream method), 33
step_into() (pfp.interp.PfpInterp method), 25, 29
step_over() (pfp.interp.PfpInterp method), 25, 29
StratGroup (class in pfp.fuzz.strats), 19
StratGroupMeta (class in pfp.fuzz.strats), 20
STRATS (in module pfp.fuzz.strats), 19
String (class in pfp.fields), 15
Struct (class in pfp.fields), 14, 15
StructUnionTypeRef() (in module pfp.interp), 26, 30

T
tell() (pfp.bitwrap.BitwrappedStream method), 33
tell_bits() (pfp.bitwrap.BitwrappedStream method), 33

U
UChar (class in pfp.fields), 15
UInt (class in pfp.fields), 16
UInt64 (class in pfp.fields), 16
unconsumed_ranges() (pfp.bitwrap.BitwrappedStream method), 33
Union (class in pfp.fields), 16
unpack_gzip() (in module pfp.native.packers), 11
UShort (class in pfp.fields), 16

V
Void (class in pfp.fields), 16

W
watch_crc() (in module pfp.native.watchers), 9
watch_length() (in module pfp.native.watchers), 9
WChar (class in pfp.fields), 16
which() (pfp.fuzz.strats.StratGroup method), 20
width (pfp.fields.Array attribute), 13, 14
write() (pfp.bitwrap.BitwrappedStream method), 33
write_bits() (pfp.bitwrap.BitwrappedStream method), 34
write_bits() (pfp.fields.BitfieldRW method), 15
WString (class in pfp.fields), 16
WUChar (class in pfp.fields), 16