PyRADS

Documentation

Release 0.1.0

Michael R. Shannon

Aug 22, 2019

CONTENTS

1 User Guide **3**

2 API Documentation **5**

2.1 rads . . . 5
2.1.1 Subpackages . . . 5
rads.rpn . . . 5
Exceptions . . . 5
Classes . . . 5
Functions . . . 5
Constants . . . 6
Operators . . . 6
rads.config . . . 18
Phase Nodes . . . 21
Variable Nodes . . . 22
rads.exceptions . . . 26
2.1.2 Configuration Loading . . . 27
2.1.3 Constants . . . 27

3 Contributor Guide **29**

3.1 Private API . . . 29
3.1.1 rads package . . . 29
Subpackages . . . 29
rads.config package . . . 29
Submodules . . . 29
rads.config.ast module . . . 29
rads.config.builders module . . . 29
rads.config.grammar module . . . 34
rads.config.loader module . . . 34
rads.config.text_parsers module . . . 34
rads.config.tree module . . . 39
rads.config.utility module . . . 48
rads.config.xml_parsers module . . . 48
Exceptions . . . 48
Parser Combinators (class based API) . . . 48
Parser Combinators (function based API) . . . 49
Module contents . . . 61
rads.xml package . . . 69
Submodules . . . 69
rads.xml.base module . . . 69
rads.xml.etree module . . . 70

rads.xml.lxml module . . . 72
rads.xml.utility module . . . 74
Module contents . . . 76
Submodules . . . 79
rads.__version__ module . . . 79
rads.constants module . . . 79
rads.datetime64util module . . . 79
rads.exceptions module . . . 80
rads.rpn module . . . 81
Exceptions . . . 81
Classes . . . 81
Functions . . . 81
Constants . . . 82
Operators . . . 82
rads.typing module . . . 94
rads.utility module . . . 94
Module contents . . . 96

4 Indices and Tables **97**

Python Module Index **99**

Index **101**

Get this documentation as a download in several formats:

- PDF
- HTMLZip
- Epub

CHAPTER

ONE

USER GUIDE

Work in progress, see the README for now.

CHAPTER

TWO

API DOCUMENTATION

If you are looking for information on a specific function, class, or method, this part of the documentation is for you, in particular *rads*.

2.1 rads

This part of the documentation covers the public classes and functions.

2.1.1 Subpackages

rads.rpn

Reverse Polish Notation calculator.

Exceptions

- *StackUnderflowError*

Classes

- *Expression*
- *Token*
- *Literal*
- *Variable*
- *Operator*

Functions

- *token()*

Constants

Keyword	Value
PI	3.141592653589793
E	2.718281828459045

Operators

Keyword	Description
SUB	a = x - y
ADD	a = x + y
MUL	a = x*y
POP	remove top of stack
NEG	a = -x
ABS	a = |x|
INV	a = 1/x
SQRT	a = sqrt(x)
SQR	a = x*x
EXP	a = exp(x)
LOG	a = ln(x)
LOG10	a = log10(x)
SIN	a = sin(x)
COS	a = cos(x)
TAN	a = tan(x)
SIND	a = sin(x) [x in degrees]
COSD	a = cos(x) [x in degrees]
TAND	a = tan(x) [x in degrees]
SINH	a = sinh(x)
COSH	a = cosh(x)
TANH	a = tanh(x)
ASIN	a = arcsin(x)
ACOS	a = arccos(x)
ATAN	a = arctan(x)
ASIND	a = arcsin(x) [a in degrees]
ACOSD	a = arccos(x) [a in degrees]
ATAND	a = arctan(x) [a in degrees]
ASINH	a = arcsinh(x)
ACOSH	a = arccosh(x)
ATANH	a = arctanh(x)
ISNAN	a = 1 if x is NaN; a = 0 otherwise
ISAN	a = 0 if x is NaN; a = 1 otherwise
RINT	a is nearest integer to x
NINT	a is nearest integer to x
CEIL	a is nearest integer greater or equal to x
CEILING	a is nearest integer greater or equal to x
FLOOR	a is nearest integer less or equal to x
D2R	convert x from degrees to radians
R2D	convert x from radian to degrees

Continued on next page

Table 1 – continued from previous page

Keyword	Description
YMDHMS	convert from seconds since 1985 to YYMMDDHHMMSS format (float)
SUM	a[i] = x[1] + … + x[i] while skipping all NaN
DIF	a[i] = x[i]-x[i-1]; a[1] = NaN
DUP	duplicate the last item on the stack
DIV	a = x/y
POW	a = x**y
FMOD	a = x modulo y
MIN	a = the lesser of x and y [element wise]
MAX	a = the greater of x and y [element wise]
ATAN2	a = arctan2(x, y)
HYPOT	a = sqrt(x*x+y*y)
R2	a = x*x + y*y
EQ	a = 1 if x == y; a = 0 otherwise
NE	a = 0 if x == y; a = 1 otherwise
LT	a = 1 if x < y; a = 0 otherwise
LE	a = 1 if x ≤ y; a = 0 otherwise
GT	a = 1 if x > y; a = 0 otherwise
GE	a = 1 if x ≥ y; a = 0 otherwise
NAN	a = NaN if x == y; a = x otherwise
AND	a = y if x is NaN; a = x otherwise
OR	a = NaN if y is NaN; a = x otherwise
IAND	a = bitwise AND of x and y
IOR	a = bitwise OR of x and y
BTEST	a = 1 if bit y of x is set; a = 0 otherwise
AVG	a = 0.5*(x+y) [when x or y is NaN a returns the other value]
DXDY	a[i] = (x[i+1]-x[i-1])/(y[i+1]-y[i-1]); a[1] = a[n] = NaN
EXCH	exchange the top two stack elements
INRANGE	a = 1 if x is between y and z (inclusive); a = 0 otherwise
BOXCAR	filter x along dimension y with boxcar of length z
GAUSS	filter x along dimension y with Gaussian of width sigma z

exception rads.rpn.**StackUnderflowError**

Bases: Exception

Raised when the stack is too small for the operation.

When this is raised the stack will exist in the state that it was before the operation was attempted. Therefore, it is not necessary to repair the stack.

class rads.rpn.**Token**

Bases: abc.ABC

Base class of all RPN tokens.

See also:

Literal A literal numeric/array value.

Variable A variable to be looked up from the environment.

Operator Base class of operators that modify the stack.

abstract property pops

Elements removed off the stack by calling the token.

abstract property puts
Elements placed on the stack by calling the token.

abstract __call__(*stack: MutableSequence[Union[int, float, bool, numpy.generic, numpy.ndarray]], environment: Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]*) → None
Perform token's action on the given *stack*.

The actions currently supported are:

- Place literal value on the stack.
- Place variable on the stack from the *environment*.
- Perform operation on the stack.
- Any combination of the above.

Note: This must be overridden for all tokens.

Parameters

- **stack** – The stack of numbers/arrays to operate on.
- **environment** – The dictionary like object providing the immutable environment.

class rads.rpn.**Literal**(*value: Union[int, float, bool]*)
Bases: *rads.rpn.Token*

Literal value token.

Parameters **value** – Value of the literal.

Raises **ValueError** – If *value* is not a number.

property pops
Elements removed off the stack by calling the token.

property puts
Elements placed on the stack by calling the token.

property value
Value of the literal.

__call__(*stack: MutableSequence[Union[int, float, bool, numpy.generic, numpy.ndarray]], environment: Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]*) → None
Place literal value on top of the given *stack*.

Parameters

- **stack** – The stack of numbers/arrays to place the value on.
- **environment** – The dictionary like object providing the immutable environment. Not used by this method.

class rads.rpn.**Variable**(*name: str*)
Bases: *rads.rpn.Token*

Environment variable token.

This is a place holder to lookup and place a number/array from an environment mapping onto the stack.

Parameters **name** – Name of the variable, this is what will be used to lookup the variables value in the environment mapping.

property pops
Elements removed off the stack by calling the token.

property puts
Elements placed on the stack by calling the token.

property name
Name of the variable, used to lookup value in the environment.

__call__(*stack: MutableSequence[Union[int, float, bool, numpy.generic, numpy.ndarray]], environment: Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]*) → None
Get variable value from *environment* and place on stack.

Parameters

- **stack** – The stack of numbers/arrays to place value on.
- **environment** – The dictionary like object to lookup the variable's value from.

Raises **KeyError** – If the variable cannot be found in the given *environment*.

class rads.rpn.**Operator**(*name: str*)
Bases: *rads.rpn.Token*, abc.ABC

Base class of all RPN operators.

Parameters **name** – Name of the operator.

class rads.rpn.**Expression**(*tokens: Union[str, Iterable[Union[int, float, bool, str, rads.rpn.Token]]]*)
Bases: collections.abc.Sequence, typing.Generic, *rads.rpn.Token*

Reverse Polish Notation expression.

Note: *Expression*s cannot be evaluated as they may not be syntactically correct. For evaluation *CompleteExpression*s are required.

Expressions can be used in three ways:

- Can be converted to a *CompleteExpression* if *pops* and *puts* are both 1 with the *complete()* method.
- Can be added to the end of a *CompleteExpression* producing a *CompleteExpression* if the *Expression* has *pops* = 1 and *puts* = 1 or a *Expression* otherwise.
- Can be added to the end of an *Expression* producing a *CompleteExpression* if the combination produces an expression with *pops* = 0 and *puts* = 1.
- Can be used as a *Token* in another expression.

See also:

CompleteExpression For a expression that can be evaluated on it's own.

Parameters **tokens** – A Reverse Polish Notation expression given as a sequence of tokens or a string of tokens.

Note: This parameter is very forgiving. If given a sequence of tokens and some of the elements are not *Token*s then an attempt will be made to convert them to *Token*s. Because of this both numbers and strings can be given in the sequence of *tokens*.

---

pops() → int
Elements removed off the stack by calling the token.

puts() → int
Elements placed on the stack by calling the token.

variables() → AbstractSet[str]
Set of variables needed to evaluate the expression.

complete() → rads.rpn.CompleteExpression
Upgrade to a *CompleteExpression* if possible.

Returns A complete expression, assuming this partial expression takes zero inputs and provides one output.

Raises **ValueError** – If the partial expression is not a valid expression.

is_complete() → bool
Determine if can be upgraded to *CompleteExpression*.

Returns True if this expression can be upgraded to a *CompleteExpression* without error with the *complete()* method.

__call__(*stack: MutableSequence[Union[int, float, bool, numpy.generic, numpy.ndarray]], environment: Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]*) → None
Evaluate the expression as a token on the given *stack*.

Parameters

- **stack** – The stack of numbers/arrays to operate on.
- **environment** – The dictionary like object providing the immutable environment.

Raises ***StackUnderflowError*** – If the expression underflows the stack.

__add__(*other: Any*) → rads.rpn.Expression

class rads.rpn.**CompleteExpression**(*tokens: Union[str, Iterable[Union[int, float, bool, str, rads.rpn.Token]]]*)
Bases: *rads.rpn.Expression*

Reverse Polish Notation expression that can be evaluated.

Parameters **tokens** – A Reverse Polish Notation expression given as a sequence of tokens or a string of tokens.

Note: This parameter is very forgiving. If given a sequence of tokens and some of the elements are not *Token*s then then an attempt will be made to convert them to *Token*s. Because of this both numbers and strings can be given in the sequence of *tokens*.

Raises **ValueError** – If the sequence or string of *tokens* represents an invalid expression. This exception also indicates which token makes the expression invalid.

complete() → rads.rpn.CompleteExpression
Return this expression as it is already complete.

Returns This complete expression.

eval(*environment: Optional[Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]] = None*) → Union[int, float, bool, numpy.generic, numpy.ndarray]
Evaluate the expression and return a numerical or logical result.

Parameters **environment** – A mapping to lookup variables in when evaluating the expression. If not provided an empty mapping will be used, this is fine as long as the expression does not contain any variables. This can be ascertained by checking the with the `variables` attribute:

```
if not expression.variables:
    expression.eval()
```

If the evaluation is lengthy or there are side effects to key lookup in the *environment* it may be beneficial to check for any missing variables first:

```
missing_vars = expression.variables.difference(environment)
```

Returns The numeric or logical result of the expression.

Raises

- **TypeError** – If there is a type mismatch with one of the operators and a value.

  Note: While this class includes a static syntax checker that runs upon initialization it does not know the type of variables in the given *environment* ahead of time.

- **KeyError** – If the expression contains a variable that is not within the given *environment*.
- **IndexError**, **ValueError**, **RuntimeError**, **ZeroDivisionError** – If arguments to operators in the expression do not have the proper dimensions or values for the operators to produce a result. See the documentation of each operator for specifics.

rads.rpn.**token**(*string: str*) → rads.rpn.Token

Parse string token into a *Token*.

There are three types of tokens that can result from this function:

- *Literal* - a literal integer or float
- *Variable* - a variable to looked up in the environment
- *Operator* - an operator to modify the stack

Parameters **string** – String to parse into a *Token*.

Returns Parsed token.

Raises

- **TypeError** – If not given a string.
- **ValueError** – If *string* is not a valid token.

rads.rpn.**SUB = SUB**

Subtract one number/array from another.

x y SUB a a = x - y

rads.rpn.**ADD = ADD**

Add two numbers/arrays.

x y ADD a a = x + y

rads.rpn.**MUL = MUL**

Multiply two numbers/arrays.

x y MUL a a = x*y

rads.rpn.**POP = POP**
Remove top of stack.

x POP remove last item from stack

rads.rpn.**NEG = NEG**
Negate number/array.

x NEG a a = -x

rads.rpn.**ABS = ABS**
Absolute value of number/array.

x ABS a a = |x|

rads.rpn.**INV = INV**
Invert number/array.

x INV a a = 1/x

rads.rpn.**SQRT = SQRT**
Compute square root of number/array.

x SQRT a a = sqrt(x)

rads.rpn.**SQR = SQR**
Square number/array.

x SQR a a = x*x

rads.rpn.**EXP = EXP**
Exponential of number/array.

x EXP a a = exp(x)

rads.rpn.**LOG = LOG**
Natural logarithm of number/array.

x LOG a a = ln(x)

rads.rpn.**LOG10 = LOG10**
Compute base 10 logarithm of number/array.

x LOG10 a a = log10(x)

rads.rpn.**SIN = SIN**
Sine of number/array [in radians].

x SIN a a = sin(x)

rads.rpn.**COS = COS**
Cosine of number/array [in radians].

x COS a a = cos(x)

rads.rpn.**TAN = TAN**
Tangent of number/array [in radians].

x TAN a a = tan(x)

rads.rpn.**SIND = SIND**
Sine of number/array [in degrees].

x SIND a a = sin(x) [x in degrees]

rads.rpn.**COSD = COSD**
Cosine of number/array [in degrees].

x COSD a a = cos(x) [x in degrees]

rads.rpn.**TAND = TAND**
Tangent of number/array [in degrees].

x TAND a a = tan(x) [x in degrees]

rads.rpn.**SINH = SINH**
Hyperbolic sine of number/array.

x SINH a a = sinh(x)

rads.rpn.**COSH = COSH**
Hyperbolic cosine of number/array.

x COSH a a = cosh(x)

rads.rpn.**TANH = TANH**
Hyperbolic tangent of number/array.

x TANH a a = tanh(x)

rads.rpn.**ASIN = ASIN**
Inverse sine of number/array [in radians].

x ASIN a a = arcsin(x)

rads.rpn.**ACOS = ACOS**
Inverse cosine of number/array [in radians].

x ACOS a a = arccos(x)

rads.rpn.**ATAN = ATAN**
Inverse tangent of number/array [in radians].

x ATAN a a = arctan(x)

rads.rpn.**ASIND = ASIND**
Inverse sine of number/array [in degrees].

x ASIND a a = arcsin(x) [a in degrees]

rads.rpn.**ACOSD = ACOSD**
Inverse cosine of number/array [in degrees].

x ACOSD a a = arccos(x) [a in degrees]

rads.rpn.**ATAND = ATAND**
Inverse tangent of number/array [in degrees].

x ATAND a a = arctan(x) [a in degrees]

rads.rpn.**ASINH = ASINH**
Inverse hyperbolic sine of number/array.

x ASINH a a = arcsinh(x)

rads.rpn.**ACOSH = ACOSH**
Inverse hyperbolic cosine of number/array.

x ACOSH a a = arccosh(x)

rads.rpn.**ATANH = ATANH**
Inverse hyperbolic tangent of number/array.

x ATANH a a = arctanh(x)

rads.rpn.**ISNAN = ISNAN**
Determine if number/array is NaN.

x ISNAN a a = 1 if x is NaN; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

rads.rpn.**ISAN = ISAN**
Determine if number/array is not NaN.

x ISAN a a = 0 if x is NaN; a = 1 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

rads.rpn.**RINT = RINT**
Round number/array to nearest integer.

x RINT a a is nearest integer to x

rads.rpn.**NINT = NINT**
Round number/array to nearest integer.

x NINT a a is nearest integer to x

rads.rpn.**CEIL = CEIL**
Round number/array up to nearest integer.

x CEIL a a is nearest integer greater or equal to x

rads.rpn.**CEILING = CEILING**
Round number/array up to nearest integer.

x CEILING a a is nearest integer greater or equal to x

rads.rpn.**FLOOR = FLOOR**
Round number/array down to nearest integer.

x FLOOR a a is nearest integer less or equal to x

rads.rpn.**D2R = D2R**
Convert number/array from degrees to radians.

x D2R a convert x from degrees to radians

rads.rpn.**R2D = R2D**
Convert number/array from radians to degrees.

x R2D a convert x from radian to degrees

rads.rpn.**YMDHMS = YMDHMS**
Convert number/array from seconds since RADS epoch to YYMMDDHHMMSS.

x YMDHMS a convert seconds of 1985 to format YYMMDDHHMMSS

Note: The top of the stack should be in seconds since the RADS epoch which is currently 1985-01-01 00:00:00 UTC

Note: The RADS documentation says this format uses a 4 digit year, but RADS uses a 2 digit year so that is what is used here.

rads.rpn.**SUM = SUM**
Compute sum over number/array [ignoring NaNs].

x SUM a a[i] = x[1] + … + x[i] while skipping all NaN

rads.rpn.**DIF = DIF**
Compute difference over number/array.

x DIF a a[i] = x[i]-x[i-1]; a[1] = NaN

rads.rpn.**DUP = DUP**
Duplicate top of stack.

x DUP a b duplicate the last item on the stack

Note: This is duplication by reference, no copy is made.

rads.rpn.**DIV = DIV**
Divide one number/array from another.

x y DIV a a = x/y

rads.rpn.**POW = POW**
Raise a number/array to the power of another number/array.

x y POW a a = x**y

rads.rpn.**FMOD = FMOD**
Remainder of dividing one number/array by another.

x y FMOD a a = x modulo y

rads.rpn.**MIN = MIN**
Minimum of two numbers/arrays [element wise].

x y MIN a a = the lesser of x and y

rads.rpn.**MAX = MAX**
Maximum of two numbers/arrays [element wise].

x y MAX a a = the greater of x and y

rads.rpn.**ATAN2 = ATAN2**
Inverse tangent of two numbers/arrays giving x and y.

x y ATAN2 a a = arctan2(x, y)

rads.rpn.**HYPOT = HYPOT**
Hypotenuse from numbers/arrays giving legs.

x y HYPOT a a = sqrt(x*x+y*y)

rads.rpn.**R2 = R2**
Sum of squares of two numbers/arrays.

x y R2 a a = x*x + y*y

rads.rpn.**EQ = EQ**
Compare two numbers/arrays for equality [element wise].

x y EQ a a = 1 if x == y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**NE = NE**
Compare two numbers/arrays for inequality [element wise].

x y NE a a = 0 if x == y; a = 1 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**LT = LT**
Compare two numbers/arrays with < [element wise].

x y LT a a = 1 if x < y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**LE = LE**
Compare two numbers/arrays with <= [element wise].

x y LE a a = 1 if x ≤ y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**GT = GT**
Compare two numbers/arrays with > [element wise].

x y GT a a = 1 if x > y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**GE = GE**
Compare two numbers/arrays with >= [element wise].

x y GE a a = 1 if x ≥ y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**NAN = NAN**
Replace number/array with NaN where it is equal to another.

x y NAN a a = NaN if x == y; a = x otherwise

rads.rpn.**AND = AND**
Fallback to second number/array when first is NaN [element wise].

x y AND a a = y if x is NaN; a = x otherwise

rads.rpn.**OR = OR**
Replace number/array with NaN where second is NaN.

x y OR a a = NaN if y is NaN; a = x otherwise

rads.rpn.**IAND = IAND**
Bitwise AND of two numbers/arrays [element wise].

x y IAND a a = bitwise AND of x and y

rads.rpn.**IOR = IOR**
Bitwise OR of two numbers/arrays [element wise].

x y IOR a a = bitwise OR of x and y

rads.rpn.**BTEST = BTEST**
Test bit, given by second number/array, in first [element wise].

x y BTEST a a = 1 if bit y of x is set; a = 0 otherwise

rads.rpn.**AVG = AVG**
Average of two numbers/arrays ignoring NaNs [element wise].

x y AVG a a = 0.5*(x+y) [when x or y is NaN a returns the other value]

rads.rpn.**DXDY = DXDY**
Compute dx/dy from two numbers/arrays.

x y DXDY a a[i] = (x[i+1]-x[i-1])/(y[i+1]-y[i-1]); a[1] = a[n] = NaN

rads.rpn.**EXCH = EXCH**
Exchange top two elements of stack.

x y EXCH a b exchange the last two items on the stack (NaNs have no influence)

rads.rpn.**INRANGE = INRANGE**
Determine if number/array is between two numbers [element wise].

x y z INRANGE a a = 1 if x is between y and z (inclusive) a = 0 otherwise (also in case of any NaN)

rads.rpn.**BOXCAR = BOXCAR**
Filter number/array with a boxcar filter along a given dimension.

x y z BOXCAR a a = filter x along monotonic dimension y with boxcar of length z (NaNs are skipped)

Note: This may behave slightly differently than the official RADS software at boundaries and at NaN values.

Raises

- **IndexError** – If x does not have dimension y.
- **ValueError** – If y or z is not a scalar.

rads.rpn.**GAUSS = GAUSS**
Filter number/array with a gaussian filter along a given dimension.

x y z GAUSS a a = filter x along monotonic dimension y with Gauss function with sigma z (NaNs are skipped)

Raises

- **IndexError** – If x does not have dimension y.
- **ValueError** – If y or z is not a scalar.

rads.config

These dataclasses make up the configuration tree returned by `rads.load_config()`. They are documented here to aid in modification of the returned configuration or for scratch construction of a *rads.config.Config* object.

class rads.config.**Config**(*pre_config: rads.config.tree.PreConfig, satellites: Mapping[str, rads.config.tree.Satellite]*)

dataclass: PyRADS configuration.

Parameters

- **pre_config** – The pre-configuration object to use when loading this configuration object.
- **satellites** – A mapping of 2 character satellite names to satellite descriptor objects.

dataroot
Path to the RADS data root.

config_files
Paths to the XML configuration files used to load this configuration.

The order is the same as they were loaded.

satellites
Mapping from 2 character satellite ID's to satellite descriptors.

See *Satellite*.

full_string() → str
Get full human friendly string representation.

Unlike `__str__()` this prints the full representation of the satellites.

Returns Human readable string representation of the PyRADS configuration.

class rads.config.**Satellite**(*id: str, id3: str, name: str, names: Sequence[str], dt1hz: float, inclination: float, frequency: Sequence[float], phases: Sequence[rads.config.tree.Phase] = <factory>, aliases: Mapping[str, Sequence[str]] = <factory>, variables: Mapping[str, rads.config.tree.Variable[typing.Union[int, float, bool]][Union[int, float, bool]]] = <factory>*)

dataclass: Satellite descriptor.

id
2 character satellite ID.

id3
3 character satellite ID.

name
Satellite name.

Note: While PyRADS places no restrictions on the length of this field to maintain compatibility with RADS it should be no longer than 8 characters.

---

names
Alternate satellite names.

dt1hz
Time step of 1-Hz data (in seconds).

inclination
Orbital inclination in degrees.

frequency
List of altimeter frequencies.

phases
Mapping from 1 character phase ID's to lists of mission phases.

Note: This being a mapping to a list of mission phases is a necessary evil brought about by satellites such as Sentinel-3B which change orbit during a mission phase.

See *Phase*.

aliases
Mapping from pseudo variables to a list of RADS variables.

When the pseudo variable is accessed any of the RADS variables listed here can be used. In particular, the first one available will be used.

variables
Mapping from variable name identifiers to variable descriptors.

These are all the variables supported by the satellite.

See *Variable*.

full_string() → str
Get full human friendly string representation.

Unlike `__str__()` this prints the full representation of the phases, aliases, and variables.

Returns Human readable string representation of the configuration for the satellite.

class rads.config.**Phase**(*id: str*, *mission: str*, *cycles: rads.config.tree.Cycles*, *repeat: rads.config.tree.Repeat*, *reference_pass: rads.config.tree.ReferencePass*, *start_time: datetime.datetime*, *end_time: Optional[datetime.datetime] = None*, *subcycles: Optional[rads.config.tree.SubCycles] = None*)
dataclass: Mission phase.

id
Single letter ID of the mission phase.

mission
Descriptive name of the mission phase.

cycles
Cycle range.

See *Cycles*.

repeat
Repeat cycle (not sub cycle) information.

See *Repeat*.

reference_pass
Equator crossing reference pass.

See *ReferencePass*.

start_time
Date and time the mission phase began.

end_time = None
Date and time the mission phase ended. This is only provided for the last mission phase of a given satellite (if that satellite has been decommissioned). In all other instances it is None.

subcycles = None
Sub cycle information for satellites with sub cycles, None otherwise.

See *SubCycles*.

class rads.config.**Variable**(*id: str, name: str, data: Union[rads.config.tree.Constant, rads.rpn.CompleteExpression, rads.config.tree.Flags, rads.config.tree.Grid, rads.config.tree.NetCDFAttribute, rads.config.tree.NetCDFVariable], units: Union[cf_units.Unit, str] = cf_units.Unit, standard_name: Optional[str] = None, source: str = '', comment: str = '', flag_values: Optional[Sequence[str]] = None, flag_masks: Optional[Sequence[str]] = None, limits: Optional[rads.config.tree.Range[~N][N]] = None, plot_range: Optional[rads.config.tree.Range[~N][N]] = None, quality_flag: Optional[Sequence[str]] = None, dimensions: int = 1, format: Optional[str] = None, compress: Optional[rads.config.tree.Compress] = None, default: Union[int, float, bool, None] = None*)

dataclass: A RADS variable descriptor.

id
Name identifier of the variable.

name
Descriptive name of the variable

data
What data backs the variable.

This can be any of the following:

- *Constant* - a numeric constant
- `CompleteExpression` - a mathematical combination of other RADS variables.
- `Flags` - an integer or boolean extracted from the "flags" RADS variable.
- *Grid* - an interpolated grid (provided by an external NetCDF file)
- *NetCDFAttribute* - a NetCDF attribute in the pass file
- *NetCDFVariable* - a NetCDF variable in the pass file

units
The variable's units.

There are three units used by RADS that are not supported by `cf_units.Unit`. The following table gives the mapping:

Unit String	`cf_units.Unit`
db	`Unit("no_unit")`
decibel	`Unit("no_unit")`
yymmddhhmmss	`Unit("unknown")`

See `cf_units.Unit`.

standard_name = None
CF-1.7 compliant "standard_name".

source = ''
Documentation of the source of the variable.

comment = ''
Comment string for the variable.

flag_values = None
List of the meanings of the integers of a enumerated flag variable.

This is mutually exclusive with *flag_masks*.

flag_masks = None
List of the meanings of the bits (LSB to MSB) for a bit flag variable.

This is mutually exclusive with *flag_values*.

limits = None
Valid range of the variable.

If outside this range the variable's data is considered bad and should be masked out.

See *Range*.

plot_range = None
Recommended plotting range for the variable.

See *Range*.

quality_flag = None
List of RADS variables that when bad make this variable bad as well.

dimensions = 1
Dimensionality of the variable.

format = None
Recommended format string to use when printing the variable's value.

compress = None
Compression scheme used for the variable.

See *Compress*.

default = None
Default numerical or boolean value to use when data sources is unavailable.

Phase Nodes

class rads.config.**Cycles**(*first: int*, *last: int*)
dataclass: Cycle range 'inclusive'.

first
First cycle of the range.

last
Last cycle of the range.

class rads.config.**Repeat**(*days: float*, *passes: int*, *longitude_drift: Optional[float] = None*)
dataclass: Length of the repeat cycle.

Note: With many satellites now using non exact repeats this is of questionable use since it is frequently disconnected from numbered cycles (which are actually sub cycles).

days
Number of days in a repeat cycle.

passes
Number of passes in a repeat cycle.

longitude_drift = None
Longitudinal drift per repeat cycle.

class rads.config.**ReferencePass**(*time: datetime.datetime*, *longitude: float*, *cycle_number: int*, *pass_number: int*, *absolute_orbit_number: int = 1*)
dataclass: Reference equator crossing.

This stores information related to a reference equator crossing used to fix the satellite in time and space.

time
Equator crossing time of the reference pass in UTC.

longitude
Longitude of the equator crossing of the reference pass.

cycle_number
Cycle number of the reference pass.

pass_number
Pass number of the reference pass.

absolute_orbit_number = 1
Absolute orbit number of reference pass.

class rads.config.**SubCycles**(*lengths: Sequence[int], start: Optional[int] = None*)
dataclass: Lengths of sub cycles.

lengths
List of the number of passes for each sub cycle.

start = None
Start cycle of the sub cycle sequence. Can be None, in which case the sub cycle sequence starts with the first cycle of the phase.

Variable Nodes

class rads.config.**Constant**(*value: Union[int, float]*)
dataclass: Numerical constant for the data field.

value
The constant numerical value.

class rads.rpn.**CompleteExpression**(*tokens: Union[str, Iterable[Union[int, float, bool, str, rads.rpn.Token]]]*)
Reverse Polish Notation expression that can be evaluated.

Parameters **tokens** – A Reverse Polish Notation expression given as a sequence of tokens or a string of tokens.

Note: This parameter is very forgiving. If given a sequence of tokens and some of the elements are not *Token*s then then an attempt will be made to convert them to *Token*s. Because of this both numbers and strings can be given in the sequence of *tokens*.

Raises **ValueError** – If the sequence or string of *tokens* represents an invalid expression. This exception also indicates which token makes the expression invalid.

complete() → rads.rpn.CompleteExpression
Return this expression as it is already complete.

Returns This complete expression.

eval(*environment: Optional[Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]] = None*) → Union[int, float, bool, numpy.generic, numpy.ndarray]
Evaluate the expression and return a numerical or logical result.

Parameters **environment** – A mapping to lookup variables in when evaluating the expression. If not provided an empty mapping will be used, this is fine as long as the expression does not contain any variables. This can be ascertained by checking the with the `variables` attribute:

```
if not expression.variables:
    expression.eval()
```

If the evaluation is lengthy or there are side effects to key lookup in the *environment* it may be beneficial to check for any missing variables first:

```
missing_vars = expression.variables.difference(environment)
```

Returns The numeric or logical result of the expression.

Raises

- **TypeError** – If there is a type mismatch with one of the operators and a value.

  Note: While this class includes a static syntax checker that runs upon initialization it does not know the type of variables in the given *environment* ahead of time.

- **KeyError** – If the expression contains a variable that is not within the given *environment*.
- **IndexError**, **ValueError**, **RuntimeError**, **ZeroDivisionError** – If arguments to operators in the expression do not have the proper dimensions or values for the operators to produce a result. See the documentation of each operator for specifics.

class rads.config.**SingleBitFlag**(*bit: int*)
dataclass: A single bit flag.

This type of flag is used for extracting true/false from a given bit.

This indicates that a single bit in the "flags" RADS variable is to be used as the data for the RADS variable.

Raises

- **TypeError** – If *bit* is not an integer.
- **ValueError** – If *bit* is negative.

bit
Bit index (starting at 0) where the flag is located.

extract(*flags: Union[int, numpy.generic, numpy.ndarray]*) → Union[int, numpy.generic, numpy.ndarray]
Extract the flag value from a number or array.

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns A bool or an array of booleans which is the value of the extracted flag.

class rads.config.**MultiBitFlag**(*bit: int*, *length: int*)

dataclass: A single bit flag.

This type of flag is used for extracting true/false from a given bit.

This indicates that 2 or more continuous bits in the "flags" RADS variable are to be used as the data for the RADS variable.

Raises

- **TypeError** – If *bit* or *length* are not integers.
- **ValueError** – If *bit* is negative or *length* is less than 2.

bit

Bit index (starting at 0) where the flag is located.

length

Length of the flag in bits.

extract(*flags: Union[int, numpy.generic, numpy.ndarray]*) → Union[int, numpy.generic, numpy.ndarray]

Extract the flag value from a number or array.

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns An integer or an array of integers which is the value of the extracted flag.

class rads.config.**SurfaceType**

dataclass: Surface type flag.

This is special flag that is based on the 3, 4, and 5 bits (zero indexed) of the underlying data and results in one of the following numerical values:

- 0 - ocean
- 2 - enclosed sea or lake
- 3 - land
- 4 - continental ice

This indicates that the surface type integer (above) is to be extracted from the "flags" RADS variable and used as the data for the RADS variable.

extract(*flags: Union[int, numpy.generic, numpy.ndarray]*) → Union[int, numpy.generic, numpy.ndarray]

Extract the flag value from a number or array.

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns The surface type integer or an array of surface type integers.

class rads.config.**Grid**(*file: str*, *x: str = 'lon'*, *y: str = 'lat'*, *method: str = 'linear'*)

dataclass: Grid file for the data field.

This indicates that the value of the grid in the NetCDF file is to be interpolated to provide data for the RADS variable.

file

NetCDF file containing the grid. This file can only contain one 2-dimensional variable.

x = 'lon'

Name of the RADS variable giving the x-coordinate for interpolation.

y = 'lat'

Name of the RADS variable giving the y-coordinate for interpolation.

method = 'linear'
Interpolation method to lookup values in the grid.

The options are:

- "linear" - bilinear interpolation
- "spline" - cubic spline interpolation
- "nearest" - nearest neighbor lookup

class rads.config.**NetCDFAttribute**(*name: str*, *variable: Optional[str] = None*, *branch: Optional[str] = None*)
dataclass: NetCDF attribute for the data field.

This indicates that the value of the NetCDF attribute from the pass file is to be used as the data for the RADS variable.

name
Name of the NetCDF attribute.

variable = None
Variable that the attribute is under. None for global.

branch = None
Postfix to append to 2 character mission folder when loading the file.

Note: PyRADS supports an unlimited number of branches. However, to maintain compatibility with RADS no more than 4 should be used.

---

class rads.config.**NetCDFVariable**(*name: str*, *branch: Optional[str] = None*)
dataclass: NetCDF variable for the data field.

This indicates that the value of the NetCDF variable from the pass file is to be used as the data for the RADS variable.

name
Name of hte NetCDF variable.

branch = None
Postfix to append to 2 character mission folder when loading the file.

Note: PyRADS supports an unlimited number of branches. However, to maintain compatibility with RADS no more than 4 should be used.

---

class rads.config.**Compress**(*type: numpy.dtype*, *scale_factor: Union[int, float] = 1*, *add_offset: Union[int, float] = 0*)
dataclass: Variable compression.

This can usally be ignored by the end user, but may prove useful if extracting and saving data into another file.

To store the variable *x*:

```
x_store = ((x - add_offset) * scale_factor).astype(type)
```

To unpack the variable *x*:

```
x = (x_store/scale_factor + add_offset).astype(np.float64)
```

type
Type of stored data as a Numpy type.

scale_factor = 1
Scale factor of stored data.

add_offset = 0
Add offset of stored data.

class rads.config.**Range**(*min: N*, *max: N*)
dataclass: Numerical range (inclusive).

min
Minimum value in range.

max
Maximum value in range.

rads.exceptions

This section documents all exceptions that should be emitted from functions in the toplevel `rads` module.

Note: If a toplevel function or method raises an exception that is not documented here please submit an issue.

class rads.exceptions.**RADSError**
Base class for all public PyRADS exceptions.

class rads.exceptions.**InvalidDataroot**
Raised when the RADS dataroot is missing or invalid.

class rads.exceptions.**ConfigError**(*message: str*, *line: Optional[int] = None*, *file: Optional[str] = None*, ***, *original: Optional[Exception] = None*)
Exception raised when there is a problem loading the configuration file.

It is usually raised after another more specific exception has been caught.

Parameters

- **message** – Error message.
- **line** – Line that cause the exception, if known.
- **file** – File that caused the exception, if known.
- **original** – Optionally the original exception.

message
Error message.

line = None
Line that cause the exception, if known (None otherwise).

file = None
File that caused the exception, if known (None otherwise).

original_exception = None
Optionally the original exception (None otherwise).

2.1.2 Configuration Loading

These functions are used to load part or all of the RADS and PyRADS configuration files.

2.1.3 Constants

Note: Cross reference links in this section will always link to the private API section below due to limitations with Sphinx.

CHAPTER

THREE

CONTRIBUTOR GUIDE

3.1 Private API

The documentation for the private API is automatically generated by *sphinx-apidoc* and is only to be used for debug and development purposes. None of the features documented here are intended for the end user. Only features documented in the *rads* are considered stable and suitable for use outside of PyRADS.

3.1.1 rads package

Subpackages

rads.config package

Submodules

rads.config.ast module

rads.config.builders module

Builders for configuration dataclasses in `rads.config.tree`.

class rads.config.builders.**SatelliteBuilder**(*, *id: str = REQUIRED*, *id3: str = REQUIRED*, *name: str = REQUIRED*, *names: Sequence[str] = REQUIRED*, *dt1hz: float = REQUIRED*, *inclination: float = REQUIRED*, *frequency: Sequence[float] = REQUIRED*, *phases: Sequence[rads.config.tree.Phase] = OPTIONAL*, *aliases: Mapping[str, Sequence[str]] = OPTIONAL*, *variables: Mapping[str, rads.config.tree.Variable[typing.Union[int, float, bool]][Union[int, float, bool]]] = OPTIONAL*)

Bases: `object`

Builder for the `rads.config.tree.Satellite` dataclass.

This class allows the `rads.config.tree.Satellite` dataclass to be constructed with the builder pattern. Once an instance is constructed simply assign to it's attributes, which are identical to the `rads.config.tree.Satellite` dataclass. When done use it's *build* method, or the `build()` function if one of the fields is *build*, to make an instance of the `rads.config.tree.Satellite` dataclass using the field values set on this builder.

Warning: Because this class overrides attribute assignment, care must be taken when extending to only use private and/or "dunder" attributes and methods.

See *rads.config.tree.Satellite* for further information on each filed.

Parameters

- **id** – Optionally initialize *id* field.
- **id3** – Optionally initialize *id3* field.
- **name** – Optionally initialize *name* field.
- **names** – Optionally initialize *names* field.
- **dt1hz** – Optionally initialize *dt1hz* field.
- **inclination** – Optionally initialize *inclination* field.
- **frequency** – Optionally initialize *frequency* field.
- **phases** – Optionally initialize *phases* field.
- **aliases** – Optionally initialize *aliases* field.
- **variables** – Optionally initialize *variables* field.

Raises

- **dataclass_builder.exceptions.UndefinedFieldError** – If you try to assign to a field that is not part of *rads.config.tree.Satellite*'s *__init__* method.
- **dataclass_builder.exceptions.MissingFieldError** – If *build()* is called on this builder before all non default fields of the dataclass are assigned.

build() → rads.config.tree.Satellite

Build a *rads.config.tree.Satellite* dataclass using the fields from this builder.

Returns An instance of the *rads.config.tree.Satellite* dataclass using the fields set on this builder instance.

Raises **dataclass_builder.exceptions.MissingFieldError** – If not all of the required fields have been assigned to this builder instance.

fields(*required: bool = True*, *optional: bool = True*) → Mapping[str, Field[Any]]

Get a dictionary of the builder's fields.

Parameters

- **required** – Set to False to not report required fields.
- **optional** – Set to False to not report optional fields.

Returns A mapping from field names to actual `dataclasses.Field`'s in the same order as in the *rads.config.tree.Satellite* dataclass.

class rads.config.builders.**PhaseBuilder**(*, *id: str = REQUIRED*, *mission: str = REQUIRED*, *cycles: rads.config.tree.Cycles = REQUIRED*, *repeat: rads.config.tree.Repeat = REQUIRED*, *reference_pass: rads.config.tree.ReferencePass = REQUIRED*, *start_time: datetime.datetime = REQUIRED*, *end_time: Optional[datetime.datetime] = OPTIONAL*, *subcycles: Optional[rads.config.tree.SubCycles] = OPTIONAL*)

Bases: object

Builder for the *rads.config.tree.Phase* dataclass.

This class allows the *rads.config.tree.Phase* dataclass to be constructed with the builder pattern. Once an instance is constructed simply assign to it's attributes, which are identical to the *rads.config.tree.Phase* dataclass. When done use it's *build* method, or the *build()* function if one of the fields is *build*, to make an instance of the *rads.config.tree.Phase* dataclass using the field values set on this builder.

Warning: Because this class overrides attribute assignment, care must be taken when extending to only use private and/or "dunder" attributes and methods.

See *rads.config.tree.Phase* for further information on each filed.

Parameters

- **id** – Optionally initialize *id* field.
- **mission** – Optionally initialize *mission* field.
- **cycles** – Optionally initialize *cycles* field.
- **repeat** – Optionally initialize *repeat* field.
- **reference_pass** – Optionally initialize *reference_pass* field.
- **start_time** – Optionally initialize *start_time* field.
- **end_time** – Optionally initialize *end_time* field.
- **subcycles** – Optionally initialize *subcycles* field.

Raises

- **dataclass_builder.exceptions.UndefinedFieldError** – If you try to assign to a field that is not part of *rads.config.tree.Phase*'s *__init__* method.
- **dataclass_builder.exceptions.MissingFieldError** – If *build()* is called on this builder before all non default fields of the dataclass are assigned.

build() → rads.config.tree.Phase

Build a *rads.config.tree.Phase* dataclass using the fields from this builder.

Returns An instance of the *rads.config.tree.Phase* dataclass using the fields set on this builder instance.

Raises **dataclass_builder.exceptions.MissingFieldError** – If not all of the required fields have been assigned to this builder instance.

fields(*required: bool = True*, *optional: bool = True*) → Mapping[str, Field[Any]]

Get a dictionary of the builder's fields.

Parameters

- **required** – Set to False to not report required fields.
- **optional** – Set to False to not report optional fields.

Returns A mapping from field names to actual dataclasses.Field's in the same order as in the *rads.config.tree.Phase* dataclass.

class rads.config.builders.**VariableBuilder**(*, *id: str = REQUIRED*, *name: str = REQUIRED*, *data: Union[rads.config.tree.Constant, rads.rpn.CompleteExpression, rads.config.tree.Flags, rads.config.tree.Grid, rads.config.tree.NetCDFAttribute, rads.config.tree.NetCDFVariable] = REQUIRED*, *units: Union[cf_units.Unit, str] = OPTIONAL*, *standard_name: Optional[str] = OPTIONAL*, *source: str = OPTIONAL*, *comment: str = OPTIONAL*, *flag_values: Optional[Sequence[str]] = OPTIONAL*, *flag_masks: Optional[Sequence[str]] = OPTIONAL*, *limits: Optional[rads.config.tree.Range[~N][N]] = OPTIONAL*, *plot_range: Optional[rads.config.tree.Range[~N][N]] = OPTIONAL*, *quality_flag: Optional[Sequence[str]] = OPTIONAL*, *dimensions: int = OPTIONAL*, *format: Optional[str] = OPTIONAL*, *compress: Optional[rads.config.tree.Compress] = OPTIONAL*, *default: Union[int, float, bool, None] = OPTIONAL*)

Bases: object

Builder for the *rads.config.tree.Variable* dataclass.

This class allows the *rads.config.tree.Variable* dataclass to be constructed with the builder pattern. Once an instance is constructed simply assign to it's attributes, which are identical to the *rads.config.tree.Variable* dataclass. When done use it's *build* method, or the *build()* function if one of the fields is *build*, to make an instance of the *rads.config.tree.Variable* dataclass using the field values set on this builder.

Warning: Because this class overrides attribute assignment, care must be taken when extending to only use private and/or "dunder" attributes and methods.

See *rads.config.tree.Variable* for further information on each filed.

Parameters

- **id** – Optionally initialize *id* field.
- **name** – Optionally initialize *name* field.
- **data** – Optionally initialize *data* field.
- **units** – Optionally initialize *units* field.
- **standard_name** – Optionally initialize *standard_name* field.
- **source** – Optionally initialize *source* field.
- **comment** – Optionally initialize *comment* field.
- **flag_values** – Optionally initialize *flag_values* field.
- **flag_masks** – Optionally initialize *flag_masks* field.
- **limits** – Optionally initialize *limits* field.
- **plot_range** – Optionally initialize *plot_range* field.

- **quality_flag** – Optionally initialize *quality_flag* field.
- **dimensions** – Optionally initialize *dimensions* field.
- **format** – Optionally initialize *format* field.
- **compress** – Optionally initialize *compress* field.
- **default** – Optionally initialize *default* field.

Raises

- **dataclass_builder.exceptions.UndefinedFieldError** – If you try to assign to a field that is not part of *rads.config.tree.Variable*'s *__init__* method.
- **dataclass_builder.exceptions.MissingFieldError** – If *build()* is called on this builder before all non default fields of the dataclass are assigned.

build() → rads.config.tree.Variable

Build a *rads.config.tree.Variable* dataclass using the fields from this builder.

Returns An instance of the *rads.config.tree.Variable* dataclass using the fields set on this builder instance.

Raises **dataclass_builder.exceptions.MissingFieldError** – If not all of the required fields have been assigned to this builder instance.

fields(*required: bool = True*, *optional: bool = True*) → Mapping[str, Field[Any]]

Get a dictionary of the builder's fields.

Parameters

- **required** – Set to False to not report required fields.
- **optional** – Set to False to not report optional fields.

Returns A mapping from field names to actual dataclasses.Field's in the same order as in the *rads.config.tree.Variable* dataclass.

class rads.config.builders.**PreConfigBuilder**(*, *dataroot: Union[str, os.PathLike] = REQUIRED*, *config_files: Sequence[Union[str, os.PathLike, IO[Any]]] = REQUIRED*, *satellites: Collection[str] = REQUIRED*, *blacklist: Collection[str] = OPTIONAL*)

Bases: object

Builder for the *rads.config.tree.PreConfig* dataclass.

This class allows the *rads.config.tree.PreConfig* dataclass to be constructed with the builder pattern. Once an instance is constructed simply assign to it's attributes, which are identical to the *rads.config.tree.PreConfig* dataclass. When done use it's *build* method, or the *build()* function if one of the fields is *build*, to make an instance of the *rads.config.tree.PreConfig* dataclass using the field values set on this builder.

Warning: Because this class overrides attribute assignment, care must be taken when extending to only use private and/or "dunder" attributes and methods.

See *rads.config.tree.PreConfig* for further information on each filed.

Parameters

- **dataroot** – Optionally initialize *dataroot* field.
- **config_files** – Optionally initialize *config_files* field.

- **satellites** – Optionally initialize *satellites* field.
- **blacklist** – Optionally initialize *blacklist* field.

Raises

- **dataclass_builder.exceptions.UndefinedFieldError** – If you try to assign to a field that is not part of *rads.config.tree.PreConfig*'s *__init__* method.
- **dataclass_builder.exceptions.MissingFieldError** – If *build()* is called on this builder before all non default fields of the dataclass are assigned.

build() → rads.config.tree.PreConfig
Build a *rads.config.tree.PreConfig* dataclass using the fields from this builder.

Returns An instance of the *rads.config.tree.PreConfig* dataclass using the fields set on this builder instance.

Raises **dataclass_builder.exceptions.MissingFieldError** – If not all of the required fields have been assigned to this builder instance.

fields(*required: bool = True*, *optional: bool = True*) → Mapping[str, Field[Any]]
Get a dictionary of the builder's fields.

Parameters

- **required** – Set to False to not report required fields.
- **optional** – Set to False to not report optional fields.

Returns A mapping from field names to actual dataclasses.Field's in the same order as in the *rads.config.tree.PreConfig* dataclass.

rads.config.grammar module

rads.config.loader module

rads.config.text_parsers module

Parsers for text within a tag.

The parsers in this file should all take a string and a mapping of attributes for the tag and return a value or take one or more parser functions and return a new parsing function.

If a parser deals with generic XML then it should be rads.config.xml_parsers and if it returns a rads.config.xml_parsers.Parser but is not generic then it belongs in rads.config.grammar.

exception rads.config.text_parsers.**TerminalTextParseError**
Bases: Exception

Raised to terminate text parsing with an error.

This error is not allowed to be handled by a text parser. It indicates that no recovery is possible.

exception rads.config.text_parsers.**TextParseError**
Bases: *rads.config.text_parsers.TerminalTextParseError*

Raised to indicate that a text parsing error has occurred.

Unlike *TerminalTextParseError* this one is allowed to be handled by a text parser.

rads.config.text_parsers.**lift**(*string_parser: Union[Callable[[str], _T], Type[_SupportsFromString]], *, terminal: bool = False*) → Callable[[str, Mapping[str, str]], _T]

Lift a simple string parser to a text parser that accepts attributes.

This is very similar to lifting a plain function into a monad.

Parameters

- **string_parser** – A simple parser that takes a string and returns a value. This can also by a type that can be constructed from a string.
- **terminal** – Set to True to use *TerminalTextParseError*s instead of *TextParseError*s.

Returns The given *string_parser* with an added argument to accept and ignore the attributes for the text tag.

Raises

- ***TextParseError*** – The resulting parser throws this if the given *string_parser* throws a TypeError, ValueError, or KeyError and *terminal* was False (the default).
- ***TerminalTextParseError*** – The resulting parser throws this if the given *string_parser* throws a TypeError, ValueError, or KeyError and *terminal* was True.

rads.config.text_parsers.**list_of**(*parser: Callable[[str, Mapping[str, str]], _T], *, sep: Optional[str] = None, terminal: bool = False*) → Callable[[str, Mapping[str, str]], List[_T]]

Convert parser into a parser of lists.

Parameters

- **parser** – Original parser.
- **sep** – Item delimiter. Default is to separate by one or more spaces.
- **terminal** – If set to True it promotes any *TextParseError* s raised by the given *parser* to a *TerminalTextParseError*.

Returns The new parser of delimited lists.

rads.config.text_parsers.**range_of**(*parser: Callable[[str, Mapping[str, str]], N], *, terminal: bool = False*) → Callable[[str, Mapping[str, str]], rads.config.tree.Range[~N][N]]

Create a range parser from a given parser for each range element.

The resulting parser will parse space separated lists of length 2 and use the given *parser* for both elements.

Parameters

- **parser** – Parser to use for the min and max values.
- **terminal** – Set to True to use *TerminalTextParseError* s instead of *TextParseError* s. Also promotes any *TextParseError* s raised by the given *parser* to a *TerminalTextParseError*.

Returns New range parser.

Raises

- ***TextParseError*** – Resulting parser raises this if given a string that does not contain two space separated elements and *terminal* was False (the default).
- ***TerminalTextParseError*** – Resulting parser raises this if given a string that does not contain two space separated elements and *terminal* was True.

rads.config.text_parsers.**one_of**(*parsers: Iterable[Callable[[str, Mapping[str, str]], Any]], *, terminal: bool = False*) → Callable[[str, Mapping[str, str]], Any]

Convert parsers into a parser that tries each one in sequence.

Note: Each parser will be tried in sequence. The next parser will be tried if *TextParseError* is raised.

Parameters

- **parsers** – A sequence of parsers the new parser should try in order.
- **terminal** – Set to True to use *TerminalTextParseError* s instead of *TextParseError* s.

Returns The new parser which tries each of the given *parsers* in order until one succeeds.

Raises

- ***TextParseError*** – Resulting parser raises this if given a string that cannot be parsed by any of the given *parsers* and *terminal* was False (the default).
- ***TerminalTextParseError*** – Resulting parser raises this if given a string that cannot be parsed by any of the given *parsers* and *terminal* was True.

rads.config.text_parsers.**compress**(*string: str, _: Mapping[str, str]*) → rads.config.tree.Compress

Parse a string into a *rads.config.tree.Compress* object.

Parameters

- **string** – String to parse into a *rads.config.tree.Compress* object. It should be in the following form:

  <type:type> [scale_factor:float] [add_offset:float]

  where only the first value is required and should be one of the following 4 character RADS data type values:

  – *int1* - maps to numpy.int8

  – *int2* - maps to numpy.int16

  – *int4* - maps to numpy.int32

  – *real* - maps to numpy.float32

  – *dble* - maps to numpy.float64

  The attribute mapping of the tag the string came from. Not currently used by this function.

- **_** – Mapping of tag attributes. Not used by this function.

Returns A new *rads.config.tree.Compress* object created from the parsed string.

Raises ***TextParseError*** – If the <type> is not in the given *string* or if too many values are in the *string*. Also, if one of the values cannot be converted.

rads.config.text_parsers.**cycles**(*string: str, _: Mapping[str, str]*) → rads.config.tree.Cycles

Parse a string into a *rads.config.tree.Cycles* object.

Parameters

- **string** – String to parse into a *rads.config.tree.Cycles* object. It should be in the following form:

```
<first cycle in phase> <last cycle in phase>
```

- **_** – Mapping of tag attributes. Not used by this function.

Returns A new *rads.config.tree.Cycles* object created from the parsed string.

Raises ***TextParseError*** – If the wrong number of values are given in the *string* or one of the values is not parsable to an integer.

rads.config.text_parsers.**data**(*string: str, attr: Mapping[str, str]*) → Any

Parse a string into one of the data objects list below.

- *rads.config.tree.Constant*
- *rads.rpn.Expression*
- *rads.config.tree.Flags*
- *rads.config.tree.Grid*
- *rads.config.tree.NetCDFAttribute*
- *rads.config.tree.NetCDFVariable*

The parsing is done based on both the given *string* and the 'source' value in *attr* if it exists.

Note: This is a terminal parser, it will either succeed or raise *TerminalTextParseError*.

Parameters

- **string** – String to parse into a data object.
- **attr** – Mapping of tag attributes. This parser can make use of the following key/value pairs if they exist:
  - "source" - explicitly specify the data source, this can be any of the following:
    - "flags"
    - "constant"
    - "grid"
    - "grid_l"
    - "grid_s"
    - "grid_c"
    - "grid_q"
    - "grid_n"
    - "nc"
    - "netcdf"
    - "math"
    - "branch" - used by some sources to specify an alternate directory
  - "x" - used by the grid sources to set the x dimension
  - "y" - used by the grid sources to set the y dimension

Returns A new data object representing the given *string*.

Raises ***TerminalTextParseError*** – If for any reason the given *string* and *attr* cannot be parsed into one of the data objects listed above.

`rads.config.text_parsers.`**`nop`**(*string: str, _: Mapping[str, str]*) → str
No operation parser, returns given string unchanged.

This exists primarily as a default for when no parser is given as the use of *lift(str)* is recommended when the parsed value is supposed to be a string.

Parameters

- **`string`** – String to return.
- **`_`** – Mapping of tag attributes. Not used by this function.

Returns The given *string*.

`rads.config.text_parsers.`**`ref_pass`**(*string: str, _: Mapping[str, str]*) → rads.config.tree.ReferencePass
Parse a string into a *`rads.config.tree.ReferencePass`* object.

Parameters

- **`string`** – String to parse into a:class:*rads.config.tree.ReferencePass* object. It should be in the following form:

```
<yyyy-mm-ddTHH:MM:SS> <lon> <cycle> <pass> [absolute orbit number]
```

  where the last element is optional and defaults to 1. The date can also be missing seconds, minutes, and hours.

- **`_`** – Mapping of tag attributes. Not used by this function.

Returns A new *`rads.config.tree.ReferencePass`* object created from the parsed string.

Raises ***TextParseError*** – If the wrong number of values are given in the *string* or one of the values is not parsable.

`rads.config.text_parsers.`**`repeat`**(*string: str, _: Mapping[str, str]*) → rads.config.tree.Repeat
Parse a string into a *`rads.config.tree.Repeat`* object.

Parameters

- **`string`** – String to parse into a *`rads.config.tree.Repeat`* object. It should be in the following form:

```
<days:float> <passes:int> [longitude drift per cycle:float]
```

  where the last value is optional.

- **`_`** – Mapping of tag attributes. Not used by this function.

Returns A new *`rads.config.tree.Repeat`* object created from the parsed string.

Raises ***TextParseError*** – If the wrong number of values are given in the *string* or one of the values is not parsable.

`rads.config.text_parsers.`**`time`**(*string: str, _: Mapping[str, str]*) → datetime.datetime
Parse a string into a `datetime.datetime` object.

Parameters

- **`string`** – String to parse into a `datetime.datetime` object. It should be in one of the following forms:

- yyyy-mm-ddTHH:MM:SS
- yyyy-mm-ddTHH:MM
- yyyy-mm-ddTHH
- yyyy-mm-ddT
- yyyy-mm-dd

- **_** – Mapping of tag attributes. Not used by this function.

Returns A new `datetime.datetime` object created from the parsed string.

Raises ***TextParseError*** – If the date/time *string* cannot be parsed.

`rads.config.text_parsers.`**`unit`**(*string: str*, *_: Mapping[str, str]*) → cf_units.Unit

Parse a string into a `cf_units.Unit` object.

Parameters

- **string** – String to parse into a `cf_units.Unit` object. See the cf_units package for supported units. If given 'dB' or 'decibel' a no_unit object will be returned and if given 'yymmddhhmmss' an unknown unit will be returned.
- **_** – Mapping of tag attributes. Not used by this function.

Returns A new `cf_units.Unit` object created from the parsed string. In the case of 'dB' and 'decibel' this will be a no_unit and in the case of 'yymmddhhmmss' it will be 'unknown'. See issue 30.

Raises **ValueError** – If the given *string* does not represent a valid unit.

rads.config.tree module

Configuration tree classes.

This module contains the classes that make up the resulting PyRADS configuration object. In particular the *rads.config.tree.Config* class.

class `rads.config.tree.`**`PreConfig`**(*dataroot: Union[str, os.PathLike]*, *config_files: Sequence[Union[str, os.PathLike, IO[Any]]]*, *satellites: Collection[str]*, *blacklist: Collection[str] = <factory>*)

Bases: `object`

dataclass: Pre configuration settings.

This is used for configuration before the individual satellite configurations are loaded.

dataroot
The location of the RADS data root.

config_files
XML configuration files used to load this pre-config. Also the XML files to use when loading the main PyRADS configuration.

satellites
A collection of 2 character satellite ID strings giving the satellites that are to be loaded. This is usually all the satellites available.

blacklist
A collection of 2 character satellite ID strings giving the satellites that should not be loaded regardless of the value of *satellites*.

class rads.config.tree.**Cycles**(*first: int*, *last: int*)
Bases: object

dataclass: Cycle range 'inclusive'.

first
First cycle of the range.

last
Last cycle of the range.

class rads.config.tree.**ReferencePass**(*time: datetime.datetime*, *longitude: float*, *cycle_number: int*, *pass_number: int*, *absolute_orbit_number: int = 1*)
Bases: object

dataclass: Reference equator crossing.

This stores information related to a reference equator crossing used to fix the satellite in time and space.

time
Equator crossing time of the reference pass in UTC.

longitude
Longitude of the equator crossing of the reference pass.

cycle_number
Cycle number of the reference pass.

pass_number
Pass number of the reference pass.

absolute_orbit_number = 1
Absolute orbit number of reference pass.

class rads.config.tree.**Repeat**(*days: float*, *passes: int*, *longitude_drift: Optional[float] = None*)
Bases: object

dataclass: Length of the repeat cycle.

Note: With many satellites now using non exact repeats this is of questionable use since it is frequently disconnected from numbered cycles (which are actually sub cycles).

---

days
Number of days in a repeat cycle.

passes
Number of passes in a repeat cycle.

longitude_drift = None
Longitudinal drift per repeat cycle.

class rads.config.tree.**SubCycles**(*lengths: Sequence[int], start: Optional[int] = None*)
Bases: object

dataclass: Lengths of sub cycles.

lengths
List of the number of passes for each sub cycle.

start = None
Start cycle of the sub cycle sequence. Can be None, in which case the sub cycle sequence starts with the first cycle of the phase.

class rads.config.tree.**Phase**(*id: str, mission: str, cycles: rads.config.tree.Cycles, repeat: rads.config.tree.Repeat, reference_pass: rads.config.tree.ReferencePass, start_time: datetime.datetime, end_time: Optional[datetime.datetime] = None, subcycles: Optional[rads.config.tree.SubCycles] = None*)

Bases: object

dataclass: Mission phase.

id
Single letter ID of the mission phase.

mission
Descriptive name of the mission phase.

cycles
Cycle range.

See *Cycles*.

repeat
Repeat cycle (not sub cycle) information.

See *Repeat*.

reference_pass
Equator crossing reference pass.

See *ReferencePass*.

start_time
Date and time the mission phase began.

end_time = None
Date and time the mission phase ended. This is only provided for the last mission phase of a given satellite (if that satellite has been decommissioned). In all other instances it is None.

subcycles = None
Sub cycle information for satellites with sub cycles, None otherwise.

See *SubCycles*.

class rads.config.tree.**Compress**(*type: numpy.dtype, scale_factor: Union[int, float] = 1, add_offset: Union[int, float] = 0*)

Bases: object

dataclass: Variable compression.

This can usally be ignored by the end user, but may prove useful if extracting and saving data into another file.

To store the variable *x*:

```
x_store = ((x - add_offset) * scale_factor).astype(type)
```

To unpack the variable *x*:

```
x = (x_store/scale_factor + add_offset).astype(np.float64)
```

type
Type of stored data as a Numpy type.

scale_factor = 1
Scale factor of stored data.

add_offset = 0
Add offset of stored data.

class rads.config.tree.**Constant**(*value: Union[int, float]*)
Bases: object

dataclass: Numerical constant for the data field.

value
The constant numerical value.

class rads.config.tree.**Flags**
Bases: abc.ABC

Base class of all data fields of type flags.

abstract extract(*flags: Union[int, float, bool, numpy.generic, numpy.ndarray]*) → Union[int, float, bool, numpy.generic, numpy.ndarray]
Extract the flag value from a number or array.

See the concrete implementations for further information:

- *SurfaceType*
- *SingleBitFlag*
- *MultiBitFlag*

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns Integer, bool, or array of integers or booleans depending on the type of flag.

class rads.config.tree.**MultiBitFlag**(*bit: int*, *length: int*)
Bases: *rads.config.tree.Flags*

dataclass: A single bit flag.

This type of flag is used for extracting true/false from a given bit.

This indicates that 2 or more continuous bits in the "flags" RADS variable are to be used as the data for the RADS variable.

Raises

- **TypeError** – If *bit* or *length* are not integers.
- **ValueError** – If *bit* is negative or *length* is less than 2.

bit
Bit index (starting at 0) where the flag is located.

length
Length of the flag in bits.

extract(*flags: Union[int, numpy.generic, numpy.ndarray]*) → Union[int, numpy.generic, numpy.ndarray]
Extract the flag value from a number or array.

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns An integer or an array of integers which is the value of the extracted flag.

class rads.config.tree.**SingleBitFlag**(*bit: int*)
Bases: *rads.config.tree.Flags*

dataclass: A single bit flag.

This type of flag is used for extracting true/false from a given bit.

This indicates that a single bit in the "flags" RADS variable is to be used as the data for the RADS variable.

Raises

- **TypeError** – If *bit* is not an integer.
- **ValueError** – If *bit* is negative.

bit
Bit index (starting at 0) where the flag is located.

extract(*flags: Union[int, numpy.generic, numpy.ndarray]*) → Union[int, numpy.generic, numpy.ndarray]
Extract the flag value from a number or array.

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns A bool or an array of booleans which is the value of the extracted flag.

class rads.config.tree.**SurfaceType**
Bases: *rads.config.tree.Flags*

dataclass: Surface type flag.

This is special flag that is based on the 3, 4, and 5 bits (zero indexed) of the underlying data and results in one of the following numerical values:

- 0 - ocean
- 2 - enclosed sea or lake
- 3 - land
- 4 - continental ice

This indicates that the surface type integer (above) is to be extracted from the "flags" RADS variable and used as the data for the RADS variable.

extract(*flags: Union[int, numpy.generic, numpy.ndarray]*) → Union[int, numpy.generic, numpy.ndarray]
Extract the flag value from a number or array.

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns The surface type integer or an array of surface type integers.

class rads.config.tree.**Grid**(*file: str*, *x: str = 'lon'*, *y: str = 'lat'*, *method: str = 'linear'*)
Bases: object

dataclass: Grid file for the data field.

This indicates that the value of the grid in the NetCDF file is to be interpolated to provide data for the RADS variable.

file
NetCDF file containing the grid. This file can only contain one 2-dimensional variable.

x = 'lon'
Name of the RADS variable giving the x-coordinate for interpolation.

y = 'lat'
Name of the RADS variable giving the y-coordinate for interpolation.

method = 'linear'
Interpolation method to lookup values in the grid.

The options are:

- "linear" - bilinear interpolation
- "spline" - cubic spline interpolation
- "nearest" - nearest neighbor lookup

class rads.config.tree.**NetCDFAttribute**(*name: str*, *variable: Optional[str] = None*, *branch: Optional[str] = None*)
Bases: object

dataclass: NetCDF attribute for the data field.

This indicates that the value of the NetCDF attribute from the pass file is to be used as the data for the RADS variable.

name
Name of the NetCDF attribute.

variable = None
Variable that the attribute is under. None for global.

branch = None
Postfix to append to 2 character mission folder when loading the file.

Note: PyRADS supports an unlimited number of branches. However, to maintain compatibility with RADS no more than 4 should be used.

---

class rads.config.tree.**NetCDFVariable**(*name: str*, *branch: Optional[str] = None*)
Bases: object

dataclass: NetCDF variable for the data field.

This indicates that the value of the NetCDF variable from the pass file is to be used as the data for the RADS variable.

name
Name of hte NetCDF variable.

branch = None
Postfix to append to 2 character mission folder when loading the file.

Note: PyRADS supports an unlimited number of branches. However, to maintain compatibility with RADS no more than 4 should be used.

---

class rads.config.tree.**Range**(*min: N*, *max: N*)
Bases: typing.Generic

dataclass: Numerical range (inclusive).

min
Minimum value in range.

max
Maximum value in range.

class rads.config.tree.**Variable**(*id: str, name: str, data: Union[rads.config.tree.Constant, rads.rpn.CompleteExpression, rads.config.tree.Flags, rads.config.tree.Grid, rads.config.tree.NetCDFAttribute, rads.config.tree.NetCDFVariable], units: Union[cf_units.Unit, str] = cf_units.Unit, standard_name: Optional[str] = None, source: str = '', comment: str = '', flag_values: Optional[Sequence[str]] = None, flag_masks: Optional[Sequence[str]] = None, limits: Optional[rads.config.tree.Range[~N][N]] = None, plot_range: Optional[rads.config.tree.Range[~N][N]] = None, quality_flag: Optional[Sequence[str]] = None, dimensions: int = 1, format: Optional[str] = None, compress: Optional[rads.config.tree.Compress] = None, default: Union[int, float, bool, None] = None*)

Bases: typing.Generic

dataclass: A RADS variable descriptor.

id
Name identifier of the variable.

name
Descriptive name of the variable

data
What data backs the variable.

This can be any of the following:

- *Constant* - a numeric constant
- CompleteExpression - a mathematical combination of other RADS variables.
- *Flags* - an integer or boolean extracted from the "flags" RADS variable.
- *Grid* - an interpolated grid (provided by an external NetCDF file)
- *NetCDFAttribute* - a NetCDF attribute in the pass file
- *NetCDFVariable* - a NetCDF variable in the pass file

units
The variable's units.

There are three units used by RADS that are not supported by cf_units.Unit. The following table gives the mapping:

Unit String	cf_units.Unit
db	Unit("no_unit")
decibel	Unit("no_unit")
yymmddhhmmss	Unit("unknown")

See cf_units.Unit.

standard_name = None
CF-1.7 compliant "standard_name".

source = ''
Documentation of the source of the variable.

comment = ''
Comment string for the variable.

flag_values = None
List of the meanings of the integers of a enumerated flag variable.

This is mutually exclusive with *flag_masks*.

flag_masks = None
List of the meanings of the bits (LSB to MSB) for a bit flag variable.

This is mutually exclusive with *flag_values*.

limits = None
Valid range of the variable.

If outside this range the variable's data is considered bad and should be masked out.

See *Range*.

plot_range = None
Recommended plotting range for the variable.

See *Range*.

quality_flag = None
List of RADS variables that when bad make this variable bad as well.

dimensions = 1
Dimensionality of the variable.

format = None
Recommended format string to use when printing the variable's value.

compress = None
Compression scheme used for the variable.

See *Compress*.

default = None
Default numerical or boolean value to use when data sources is unavailable.

class rads.config.tree.**Satellite**(*id: str, id3: str, name: str, names: Sequence[str], dt1hz: float, inclination: float, frequency: Sequence[float], phases: Sequence[rads.config.tree.Phase] = <factory>, aliases: Mapping[str, Sequence[str]] = <factory>, variables: Mapping[str, rads.config.tree.Variable[typing.Union[int, float, bool]][Union[int, float, bool]]] = <factory>*)

Bases: object

dataclass: Satellite descriptor.

id
2 character satellite ID.

id3
3 character satellite ID.

name
Satellite name.

Note: While PyRADS places no restrictions on the length of this field to maintain compatibility with RADS it should be no longer than 8 characters.

---

names
Alternate satellite names.

dt1hz
Time step of 1-Hz data (in seconds).

inclination
Orbital inclination in degrees.

frequency
List of altimeter frequencies.

phases
Mapping from 1 character phase ID's to lists of mission phases.

Note: This being a mapping to a list of mission phases is a necessary evil brought about by satellites such as Sentinel-3B which change orbit during a mission phase.

See *Phase*.

aliases
Mapping from pseudo variables to a list of RADS variables.

When the pseudo variable is accessed any of the RADS variables listed here can be used. In particular, the first one available will be used.

variables
Mapping from variable name identifiers to variable descriptors.

These are all the variables supported by the satellite.

See *Variable*.

full_string() → str
Get full human friendly string representation.

Unlike `__str__()` this prints the full representation of the phases, aliases, and variables.

Returns Human readable string representation of the configuration for the satellite.

class rads.config.tree.**Config**(*pre_config: rads.config.tree.PreConfig, satellites: Mapping[str, rads.config.tree.Satellite]*)

Bases: object

dataclass: PyRADS configuration.

Parameters

- **pre_config** – The pre-configuration object to use when loading this configuration object.
- **satellites** – A mapping of 2 character satellite names to satellite descriptor objects.

dataroot
Path to the RADS data root.

config_files
Paths to the XML configuration files used to load this configuration.

The order is the same as they were loaded.

satellites
Mapping from 2 character satellite ID's to satellite descriptors.

See *Satellite*.

full_string() → str
Get full human friendly string representation.

Unlike __str__() this prints the full representation of the satellites.

Returns Human readable string representation of the PyRADS configuration.

rads.config.utility module

rads.config.xml_parsers module

Parser combinators for reading XML files.

This module is heavily based on PEGTL, a parser combinator library for C++.

Exceptions

Name	Description
TerminalXMLParseError	Parse error that fails parsing of XML file.
XMLParseError	Parse error that fails a single parser.

Parser Combinators (class based API)

Name	Description
Parser	Base class of all parser combinators.
Apply	Apply a function the value result of a parser.
Lazy	Delay parser construction until evaluation.
Must	Require parser to succeed.
At	Non consuming match.
Not	Invert a parser match, non consuming.
Repeat	Match zero or more times (greedily).
Sequence	Chain parsers together, all or nothing match.
Alternate	Chain parsers together, taking first success.
Success	Always succeeds, non consuming.
Failure	Always fails, non consuming.
Start	Match beginning of XML block.
End	Match end of XML block.
AnyElement	Match any XML element.
Tag	Match an XML element by it's tag name.

Parser Combinators (function based API)

Name	Description
lazy()	Delay parser construction until evaluation.
at()	Non consuming match.
not_at()	Invert a parser match, non consuming.
opt()	Optional match, always succeeds.
plus()	Match one or more times (greedily).
seq()	Match a sequence of parsers, all or nothing.
sor()	Use first matching parser.
star()	Match zero ore more times (greedily).
must()	Require parser to succeed.
rep()	Match N times.
until()	Consume elements until a match, match not consumed.
failure()	Always fails, non consuming.
success()	Always succeeds, non consuming.
start()	Match beginning of XML block.
end()	Match end of XML block.
any()	Match any XML element.
tag()	Match an XML element by it's tag name.

exception rads.config.xml_parsers.**TerminalXMLParseError**(*file: Optional[str], line: Optional[int], message: Optional[str] = None*)

Bases: Exception

A parse error that should fail parsing of the entire file.

Parameters

- **file** – Name of file that was being parsed when the error occurred.
- **line** – Line number in the *file* that was being parsed when the error occurred.
- **message** – An optional message (instead of the default 'parsing failed') detailing why the parse failed.

file = None
Filename where the error occurred.

line = None
Line number in the *file* where the error occurred.

message = 'parsing failed'
The message provided when the error was constructed.

exception rads.config.xml_parsers.**XMLParseError**(*file: Optional[str], line: Optional[int], message: Optional[str] = None*)

Bases: *rads.config.xml_parsers.TerminalXMLParseError*

A parse error that signals that a given parser failed.

Unlike *TerminalXMLParseError* is expected and simply signals that the parser did not match and another parser should be tried.

Parameters

- **file** – Name of file that was being parsed when the error occurred.

- **line** – Line number in the *file* that was being parsed when the error occurred.
- **message** – An optional message (instead of the default 'parsing failed') detailing why the parse failed.

terminal(*message: Optional[str] = None*) → rads.config.xml_parsers.TerminalXMLParseError
Raise this local failure to a global failure.

Parameters **message** – Optionally a new message.

Returns A global parse failure with the same file and line, and possibly message as this exception.

rads.config.xml_parsers.**next_element**(*pos: rads.xml.base.Element*) → rads.xml.base.Element
Get next element lazily.

Parameters **pos** – Current element.

Returns Next sibling XML element.

Raises ***XMLParseError*** – If there is no next sibling element.

rads.config.xml_parsers.**first_child**(*pos: rads.xml.base.Element*) → rads.xml.base.Element
Get first child of element, lazily.

class rads.config.xml_parsers.**Parser**
Bases: abc.ABC

Base parser combinator.

abstract **__call__**(*position: rads.xml.base.Element*) → Tuple[Any, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a yzal.Thunk and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

__add__(*other: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Sequence
Combine two parsers, matching the first followed by the second.

Multiple consecutive uses of '+' will result in a single *Sequence* because the *Sequence* class automatically flattens itself.

Parameters **other** – The parser to match after this one.

Returns A new parser that will match this parser followed by the *other* parser (if the this parser matched).

__or__(*other: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Alternate
Combine two parsers, matching the first or the second.

Multiple consecutive uses of 'l' will result in a single `Alternate` because the `Alternate` class automatically flattens itself.

Parameters **other** – The parser to match if this parser does not.

Returns A new parser that will match either this parser or the *other* parser (if the this parser did not match).

__xor__(*func: Callable[[Any], Any]*) → rads.config.xml_parsers.Apply
Apply a function to the value result of this parser.

Parameters **func** – The function to apply to the value result of matching this parser.

Note: This will not be ran until this parser is matched.

Returns A new parser that will match this parser and upon a successful match apply the given *func* to the value result.

__invert__() → rads.config.xml_parsers.Not
Invert this parser.

If this parser would match a given position, now it will not. If it would not match now it will, but it will not consume any elements.

Returns A new parser that will not match whenever this parser does, and will match whenever this parser does not. However, it will not consume any elements.

__lshift__(*message: str*) → rads.config.xml_parsers.Must
Require the parser to succeed.

This will convert all `XMLParseError` s to `TerminalXMLParseError` s.

Parameters **message** – The message that will be raised if the parser does not match.

Returns A new parser that will elevate any local parse failures to global failures and overwrite their message with *message*.

class rads.config.xml_parsers.**Apply**(*parser: rads.config.xml_parsers.Parser, func: Callable[[Any], Any], catch: Optional[Collection[type]] = None*)
Bases: `rads.config.xml_parsers.Parser`

Apply a function to the value result of the parser.

Parameters

- **parser** – The parser whose value result to apply the given *func* to the value result of.
- **func** – The function to apply.
- **catch** – An exception or iterable of exceptions to convert into `XMLParseError` s. The default is not to catch any exceptions. To catch all exceptions simply pass `Exception` as it is the base class of all exceptions that should be caught.

Note: Any exceptions that are derived from the exceptions given will also be caught.

__call__(*position: rads.xml.base.Element*) → Tuple[Any, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**Lazy**(*parser_func: Callable[[], rads.config.xml_parsers.Parser]*)

Bases: *rads.config.xml_parsers.Parser*

Delay construction of parser until evaluated.

Note: This lazy behavior is useful when constructing recursive parsers in order to avoid infinite recursion.

Parameters **parser_func** – A zero argument function that returns a parser when called. This will be used to delay construction of the parser.

__call__(*position: rads.xml.base.Element*) → Tuple[Any, rads.xml.base.Element]

Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**Must**(*parser: rads.config.xml_parsers.Parser*, *message: Optional[str] = None*)

Bases: *rads.config.xml_parsers.Parser*

Raise a XMLParseError to a TerminalXMLParseError ending parsing.

Parameters

- **parser** – Parser that must match.
- **message** – New message to apply to the `GlobalParserFailure` if the parser does not match.

__call__ (*position: rads.xml.base.Element*) → Tuple[Any, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**At** (*parser: rads.config.xml_parsers.Parser*)
Bases: *rads.config.xml_parsers.Parser*

Match a parser, consuming nothing.

Parameters **parser** – Parser to match.

__call__ (*position: rads.xml.base.Element*) → Tuple[Any, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**Not**(*parser: rads.config.xml_parsers.Parser*)
Bases: *rads.config.xml_parsers.Parser*

Invert a parser match, consuming nothing.

Parameters **parser** – Parser to invert the match of.

__call__(*position: rads.xml.base.Element*) → Tuple[None, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**Repeat**(*parser: rads.config.xml_parsers.Parser*)
Bases: *rads.config.xml_parsers.Parser*

Match a parser zero or more times (greedily).

Parameters **parser** – Parser to match repeatedly.

__call__(*position: rads.xml.base.Element*) → Tuple[List[Any], rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**Sequence**(**parsers: rads.config.xml_parsers.Parser*)
Bases: rads.config.xml_parsers._MultiParser

Chain parsers together, succeeding only if all succeed in order.

Note: Consecutive Sequence's are automatically flattened.

Parameters ***parsers** – Parsers to match in sequence.

__call__(*position: rads.xml.base.Element*) → Tuple[List[Any], rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a yzal.Thunk and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

__add__(*other: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Sequence
Combine this sequence and a parser, returning a new sequence.

Note: If the *other* parser is a *Sequence* then the parsers in the *other* *Sequence* will be unwrapped and appended individually.

Parameters **other** – The parser to combine with this sequence to form the new sequence.

Returns A new sequence which matches this sequence followed by the given parser (if the sequence matched).

__iadd__(*other: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Sequence
Combine this sequence with the given parser (in place).

Note: If the *other* parser is a *Sequence* then the parsers in the *other* *Sequence* will be unwrapped and appended to this sequence individually.

Parameters **other** – The parser to combine with (append to) this sequence.

Returns This sequence parser.

class rads.config.xml_parsers.**Alternate**(**parsers: rads.config.xml_parsers.Parser*)
Bases: rads.config.xml_parsers._MultiParser

Match any one of the parsers, stops on first match.

Note: Consecutive Alternate's are automatically flattened.

Parameters ***parsers** – Pool of parsers to find a match in.

__call__(*position: rads.xml.base.Element*) → Tuple[Any, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a yzal.Thunk and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

__or__(*other: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Alternate
Combine this alternate and a parser, returning a new alternate.

Note: If the *other* parser is a *Alternate* then the parsers in the *other* *Alternate* will be unwrapped and added individually.

Parameters **other** – The parser to combine with this alternate to form the new alternate.

Returns A new alternate which matches any parser from this alternate or the given parser (if no parser of this alternate matches).

class rads.config.xml_parsers.**Success**
Bases: *rads.config.xml_parsers.Parser*

Parser that always succeeds, consuming nothing.

__call__(*position: rads.xml.base.Element*) → Tuple[None, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**Failure**
Bases: *rads.config.xml_parsers.Parser*

Parser that always fails, consuming nothing.

__call__(*position: rads.xml.base.Element*) → NoReturn
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**Start**
Bases: *rads.config.xml_parsers.Parser*

Match start of an element, consuming nothing.

__call__(*position: rads.xml.base.Element*) → Tuple[None, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**End**
Bases: *rads.config.xml_parsers.Parser*

Match end of an element, consuming nothing.

__call__(*position: rads.xml.base.Element*) → Tuple[None, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**AnyElement**
Bases: *rads.config.xml_parsers.Parser*

Parser that matches any element.

__call__(*position: rads.xml.base.Element*) → Tuple[rads.xml.base.Element, rads.xml.base.Element]
Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

class rads.config.xml_parsers.**Tag**(*name: str*)

Bases: *rads.config.xml_parsers.Parser*

Match an element by it's tag name.

Parameters **name** – Tag name to match.

__call__(*position: rads.xml.base.Element*) → Tuple[rads.xml.base.Element, rads.xml.base.Element]

Call the parser, trying to match at the given *position*.

If the match fails a *XMLParseError* will be raised. This call will only return if the parser matches at the given *position*.

Parameters **position** – An XML element that the parser should attempt to match at.

Returns

A tuple giving the value result of the match (which depends on the particular parser) and the element to match at.

The next element can be the same element as given in *position* (a non consuming parser) or any later sibling element.

Further, it will actually be a `yzal.Thunk` and will therefore delay it's construction until it is needed. Therefore, any *XMLParseError* that may be generated by moving to a later element will occur when the returned element is used.

Raises

- ***XMLParseError*** – If the parser does not match at the given *position*.
- ***TerminalXMLParseError*** – If the parser encounters an unrecoverable error.

rads.config.xml_parsers.**lazy**(*parser_func: Callable[[], rads.config.xml_parsers.Parser]*) → rads.config.xml_parsers.Parser

Delays construction of parser until evaluated.

Parameters **parser_func** – A zero argument function that returns a parser when called. This will be used to delay construction of the parser.

Returns A new parser that is equivalent to the parser returned by *parser_func*.

rads.config.xml_parsers.**at**(*parser: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Parser

Succeeds if and only if the given parser succeeds, consumes nothing.

Parameters **parser** – The parser that must succeed.

Returns A new parser that succeeds if and only if *parser* succeeds, but does not consume input.

rads.config.xml_parsers.**not_at**(*parser: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Parser

Succeeds if and only if the given parser fails, consumes nothing.

Parameters **parser** – The parser that must fail.

Returns A new parser that succeeds if and only if *parser* fails, but does not consume input.

rads.config.xml_parsers.**opt**(*parser: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Parser

Parser that always succeeds, regardless of the given parser.

Parameters **parser** – An optional parser that can succeed or fail.

Returns A new parser that optionally matches *parser*. If *parser* succeeds this parser will be transparent, as if *parser* was called directly. If *parser* fails this `opt()` returns None as the result and does not consume anything.

rads.config.xml_parsers.**plus**(*parser: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Parser

Match the given parser as much as possible, must match at least once.

Parameters **parser** – Parser to match one or more times (greedily).

Returns A new parser that matches *parser* one or more times. Failing if no matches are made.

rads.config.xml_parsers.**seq**(**parsers: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Parser

Match sequence of parsers in order, succeeding if and only if all succeed.

Parameters ***parsers** – One or more parsers to match in order.

Returns A new parser that matches all the given *parser*'s in order, failing if any one of the *parser*'s fail.

rads.config.xml_parsers.**sor**(**parsers: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Parser

Match the first of the given parsers, failing if all fail.

Parameters ***parsers** – One or more parsers to match. The first parser that succeeds will take the place of this parser. If all fail then this parser will also fail.

Returns A new parser that matches the first *parser* that succeeds or fails if all *parser*'s fail.

rads.config.xml_parsers.**star**(*parser: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Parser

Match the given parser as much as possible, can match zero times.

Parameters **parser** – Parser to match zero or more times (greedily).

Returns A new parser that matches *parser* one or more times. Failing if no matches are made.

rads.config.xml_parsers.**must**(*parser: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Parser

Raise a local parse failure to a global parse failure.

Local parse failures (`XMLParseError`) are typically caught by `Alternate` or other such parsers that allow some parser's to fail. In particular, local failures are an expected part of parser combinators and simply signal that a particular parser could not parse the given elements. A global parse failure (`TerminalXMLParseError`) should only be caught at the top level and signals that the entire parse is a failure.

Parameters **parser** – A parser that must match, else the entire parse is failed.

Returns A parser that must succeed, if it fails a `GlobalParserFailure` is raised.

rads.config.xml_parsers.**rep**(*parser: rads.config.xml_parsers.Parser*, *times: int*) → rads.config.xml_parsers.Parser

Match the given parser a given number of times.

Fails if the parser does not succeed the given number of times.

Parameters

- **parser** – The parser to match *times*.
- **times** – Number of times the *parser* must succeed.

Returns A parser that succeeds only if the given *parser* matches the given number of *times*.

rads.config.xml_parsers.**until**(*parser: rads.config.xml_parsers.Parser*) → rads.config.xml_parsers.Parser

Match all elements until the given *parser* matches.

Does not consume the elements that the given *parser* matches.

Parameters **parser** – The parser to end matching with.

Returns A parser that will consume all elements until the given *parser* matches. It will not consume the elements that the given *parser* matched.

rads.config.xml_parsers.**failure**() → rads.config.xml_parsers.Parser

Parser that always fails.

Returns A new parser that always fails, consuming nothing.

rads.config.xml_parsers.**success**() → rads.config.xml_parsers.Parser

Parser that always succeeds.

Returns A new parser that always succeeds, consuming nothing.

rads.config.xml_parsers.**start**() → rads.config.xml_parsers.Parser

Match the beginning of an element.

Returns A new parser that matches the beginning of an element, consuming nothing.

rads.config.xml_parsers.**end**() → rads.config.xml_parsers.Parser

Match the end of an element.

Returns A new parser that matches the end of an element, consuming nothing.

rads.config.xml_parsers.**any**() → rads.config.xml_parsers.Parser

Match any element.

Returns A new parser that matches any single element.

rads.config.xml_parsers.**tag**(*name: str*) → rads.config.xml_parsers.Parser

Match an element by tag name.

Parameters **name** – Tag name to match.

Returns Parser matching the given *tag* name.

Module contents

PyRADS configuration file API.

class rads.config.**Compress**(*type: numpy.dtype*, *scale_factor: Union[int, float] = 1*, *add_offset: Union[int, float] = 0*)

Bases: object

dataclass: Variable compression.

This can usally be ignored by the end user, but may prove useful if extracting and saving data into another file.

To store the variable *x*:

```
x_store = ((x - add_offset) * scale_factor).astype(type)
```

To unpack the variable *x*:

```
x = (x_store/scale_factor + add_offset).astype(np.float64)
```

type
Type of stored data as a Numpy type.

scale_factor = 1
Scale factor of stored data.

add_offset = 0
Add offset of stored data.

class rads.config.**Config**(*pre_config: rads.config.tree.PreConfig, satellites: Mapping[str, rads.config.tree.Satellite]*)
Bases: object

dataclass: PyRADS configuration.

Parameters

- **pre_config** – The pre-configuration object to use when loading this configuration object.
- **satellites** – A mapping of 2 character satellite names to satellite descriptor objects.

dataroot
Path to the RADS data root.

config_files
Paths to the XML configuration files used to load this configuration.

The order is the same as they were loaded.

satellites
Mapping from 2 character satellite ID's to satellite descriptors.

See *Satellite*.

full_string() → str
Get full human friendly string representation.

Unlike __str__() this prints the full representation of the satellites.

Returns Human readable string representation of the PyRADS configuration.

class rads.config.**Constant**(*value: Union[int, float]*)
Bases: object

dataclass: Numerical constant for the data field.

value
The constant numerical value.

class rads.config.**Cycles**(*first: int*, *last: int*)
Bases: object

dataclass: Cycle range 'inclusive'.

first
First cycle of the range.

last
Last cycle of the range.

class rads.config.**Grid**(*file: str*, *x: str = 'lon'*, *y: str = 'lat'*, *method: str = 'linear'*)
Bases: object

dataclass: Grid file for the data field.

This indicates that the value of the grid in the NetCDF file is to be interpolated to provide data for the RADS variable.

file
NetCDF file containing the grid. This file can only contain one 2-dimensional variable.

x = 'lon'
Name of the RADS variable giving the x-coordinate for interpolation.

y = 'lat'
Name of the RADS variable giving the y-coordinate for interpolation.

method = 'linear'
Interpolation method to lookup values in the grid.

The options are:

- "linear" - bilinear interpolation
- "spline" - cubic spline interpolation
- "nearest" - nearest neighbor lookup

class rads.config.**MultiBitFlag**(*bit: int*, *length: int*)
Bases: *rads.config.tree.Flags*

dataclass: A single bit flag.

This type of flag is used for extracting true/false from a given bit.

This indicates that 2 or more continuous bits in the "flags" RADS variable are to be used as the data for the RADS variable.

Raises

- **TypeError** – If *bit* or *length* are not integers.
- **ValueError** – If *bit* is negative or *length* is less than 2.

bit
Bit index (starting at 0) where the flag is located.

length
Length of the flag in bits.

extract(*flags: Union[int, numpy.generic, numpy.ndarray]*) → Union[int, numpy.generic, numpy.ndarray]
Extract the flag value from a number or array.

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns An integer or an array of integers which is the value of the extracted flag.

class rads.config.**NetCDFAttribute**(*name: str*, *variable: Optional[str] = None*, *branch: Optional[str] = None*)
Bases: object

dataclass: NetCDF attribute for the data field.

This indicates that the value of the NetCDF attribute from the pass file is to be used as the data for the RADS variable.

name
Name of the NetCDF attribute.

variable = None
Variable that the attribute is under. None for global.

branch = None
Postfix to append to 2 character mission folder when loading the file.

Note: PyRADS supports an unlimited number of branches. However, to maintain compatibility with RADS no more than 4 should be used.

---

class rads.config.**NetCDFVariable**(*name: str*, *branch: Optional[str] = None*)
Bases: object

dataclass: NetCDF variable for the data field.

This indicates that the value of the NetCDF variable from the pass file is to be used as the data for the RADS variable.

name
Name of hte NetCDF variable.

branch = None
Postfix to append to 2 character mission folder when loading the file.

Note: PyRADS supports an unlimited number of branches. However, to maintain compatibility with RADS no more than 4 should be used.

---

class rads.config.**Phase**(*id: str*, *mission: str*, *cycles: rads.config.tree.Cycles*, *repeat: rads.config.tree.Repeat*, *reference_pass: rads.config.tree.ReferencePass*, *start_time: datetime.datetime*, *end_time: Optional[datetime.datetime] = None*, *subcycles: Optional[rads.config.tree.SubCycles] = None*)
Bases: object

dataclass: Mission phase.

id
Single letter ID of the mission phase.

mission
Descriptive name of the mission phase.

cycles
Cycle range.

See *Cycles*.

repeat
Repeat cycle (not sub cycle) information.

See *Repeat*.

reference_pass
Equator crossing reference pass.

See *ReferencePass*.

start_time
Date and time the mission phase began.

end_time = None
Date and time the mission phase ended. This is only provided for the last mission phase of a given satellite (if that satellite has been decommissioned). In all other instances it is None.

subcycles = None
Sub cycle information for satellites with sub cycles, None otherwise.

See *SubCycles*.

class rads.config.**Range**(*min: N*, *max: N*)
Bases: typing.Generic

dataclass: Numerical range (inclusive).

min
Minimum value in range.

max
Maximum value in range.

class rads.config.**ReferencePass**(*time: datetime.datetime*, *longitude: float*, *cycle_number: int*, *pass_number: int*, *absolute_orbit_number: int = 1*)
Bases: object

dataclass: Reference equator crossing.

This stores information related to a reference equator crossing used to fix the satellite in time and space.

time
Equator crossing time of the reference pass in UTC.

longitude
Longitude of the equator crossing of the reference pass.

cycle_number
Cycle number of the reference pass.

pass_number
Pass number of the reference pass.

absolute_orbit_number = 1
Absolute orbit number of reference pass.

class rads.config.**Repeat**(*days: float*, *passes: int*, *longitude_drift: Optional[float] = None*)
Bases: object

dataclass: Length of the repeat cycle.

Note: With many satellites now using non exact repeats this is of questionable use since it is frequently disconnected from numbered cycles (which are actually sub cycles).

---

days
Number of days in a repeat cycle.

passes
Number of passes in a repeat cycle.

longitude_drift = None
Longitudinal drift per repeat cycle.

class rads.config.**Satellite**(*id: str, id3: str, name: str, names: Sequence[str], dt1hz: float, inclination: float, frequency: Sequence[float], phases: Sequence[rads.config.tree.Phase] = <factory>, aliases: Mapping[str, Sequence[str]] = <factory>, variables: Mapping[str, rads.config.tree.Variable[typing.Union[int, float, bool]][Union[int, float, bool]]] = <factory>*)

Bases: object

dataclass: Satellite descriptor.

id
2 character satellite ID.

id3
3 character satellite ID.

name
Satellite name.

Note: While PyRADS places no restrictions on the length of this field to maintain compatibility with RADS it should be no longer than 8 characters.

names
Alternate satellite names.

dt1hz
Time step of 1-Hz data (in seconds).

inclination
Orbital inclination in degrees.

frequency
List of altimeter frequencies.

phases
Mapping from 1 character phase ID's to lists of mission phases.

Note: This being a mapping to a list of mission phases is a necessary evil brought about by satellites such as Sentinel-3B which change orbit during a mission phase.

See *Phase*.

aliases
Mapping from pseudo variables to a list of RADS variables.

When the pseudo variable is accessed any of the RADS variables listed here can be used. In particular, the first one available will be used.

variables
Mapping from variable name identifiers to variable descriptors.

These are all the variables supported by the satellite.

See *Variable*.

full_string() → str
Get full human friendly string representation.

Unlike __str__() this prints the full representation of the phases, aliases, and variables.

Returns Human readable string representation of the configuration for the satellite.

class rads.config.**SingleBitFlag**(*bit: int*)
Bases: *rads.config.tree.Flags*

dataclass: A single bit flag.

This type of flag is used for extracting true/false from a given bit.

This indicates that a single bit in the "flags" RADS variable is to be used as the data for the RADS variable.

Raises

- **TypeError** – If *bit* is not an integer.
- **ValueError** – If *bit* is negative.

bit
Bit index (starting at 0) where the flag is located.

extract(*flags: Union[int, numpy.generic, numpy.ndarray]*) → Union[int, numpy.generic, numpy.ndarray]
Extract the flag value from a number or array.

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns A bool or an array of booleans which is the value of the extracted flag.

class rads.config.**SubCycles**(*lengths: Sequence[int], start: Optional[int] = None*)
Bases: object

dataclass: Lengths of sub cycles.

lengths
List of the number of passes for each sub cycle.

start = None
Start cycle of the sub cycle sequence. Can be None, in which case the sub cycle sequence starts with the first cycle of the phase.

class rads.config.**SurfaceType**
Bases: *rads.config.tree.Flags*

dataclass: Surface type flag.

This is special flag that is based on the 3, 4, and 5 bits (zero indexed) of the underlying data and results in one of the following numerical values:

- 0 - ocean
- 2 - enclosed sea or lake
- 3 - land
- 4 - continental ice

This indicates that the surface type integer (above) is to be extracted from the "flags" RADS variable and used as the data for the RADS variable.

extract(*flags: Union[int, numpy.generic, numpy.ndarray]*) → Union[int, numpy.generic, numpy.ndarray]
Extract the flag value from a number or array.

Parameters **flags** – Integer or array of integers to extract flag value from.

Returns The surface type integer or an array of surface type integers.

class rads.config.**Variable**(*id: str, name: str, data: Union[rads.config.tree.Constant, rads.rpn.CompleteExpression, rads.config.tree.Flags, rads.config.tree.Grid, rads.config.tree.NetCDFAttribute, rads.config.tree.NetCDFVariable], units: Union[cf_units.Unit, str] = cf_units.Unit, standard_name: Optional[str] = None, source: str = '', comment: str = '', flag_values: Optional[Sequence[str]] = None, flag_masks: Optional[Sequence[str]] = None, limits: Optional[rads.config.tree.Range[~N][N]] = None, plot_range: Optional[rads.config.tree.Range[~N][N]] = None, quality_flag: Optional[Sequence[str]] = None, dimensions: int = 1, format: Optional[str] = None, compress: Optional[rads.config.tree.Compress] = None, default: Union[int, float, bool, None] = None*)

Bases: typing.Generic

dataclass: A RADS variable descriptor.

id
Name identifier of the variable.

name
Descriptive name of the variable

data
What data backs the variable.

This can be any of the following:

- *Constant* - a numeric constant
- CompleteExpression - a mathematical combination of other RADS variables.
- Flags - an integer or boolean extracted from the "flags" RADS variable.
- *Grid* - an interpolated grid (provided by an external NetCDF file)
- *NetCDFAttribute* - a NetCDF attribute in the pass file
- *NetCDFVariable* - a NetCDF variable in the pass file

units
The variable's units.

There are three units used by RADS that are not supported by cf_units.Unit. The following table gives the mapping:

Unit String	cf_units.Unit
db	Unit("no_unit")
decibel	Unit("no_unit")
yymmddhhmmss	Unit("unknown")

See cf_units.Unit.

standard_name = None
CF-1.7 compliant "standard_name".

source = ''
Documentation of the source of the variable.

comment = ''
Comment string for the variable.

flag_values = None
List of the meanings of the integers of a enumerated flag variable.

This is mutually exclusive with *flag_masks*.

flag_masks = None
List of the meanings of the bits (LSB to MSB) for a bit flag variable.

This is mutually exclusive with *flag_values*.

limits = None
Valid range of the variable.

If outside this range the variable's data is considered bad and should be masked out.

See *Range*.

plot_range = None
Recommended plotting range for the variable.

See *Range*.

quality_flag = None
List of RADS variables that when bad make this variable bad as well.

dimensions = 1
Dimensionality of the variable.

format = None
Recommended format string to use when printing the variable's value.

compress = None
Compression scheme used for the variable.

See *Compress*.

default = None
Default numerical or boolean value to use when data sources is unavailable.

rads.xml package

Submodules

rads.xml.base module

Generic XML tools, not relating to a specific backend.

class rads.xml.base.**Element**
Bases: collections.abc.Iterable, typing.Generic, collections.abc.Sized, abc.ABC

A generic XML element.

Base class of XML elements.

dumps(*, *indent: Union[int, str, None] = None*, *_current_indent: str = ''*) → str
Get string representation of this element and all child elements.

Parameters **indent** – Amount to indent each level. Can be given as an int or a string. Defaults to 4 spaces.

Returns String representation of this and all child elements.

abstract next() → rads.xml.base.Element
Get the next sibling element.

Returns Next XML sibling element.

Raises **StopIteration** – If there is no next sibling element.

abstract prev() → rads.xml.base.Element
Get the previous sibling element.

Returns Previous XML sibling element.

Raises **StopIteration** – If there is no previous sibling element.

abstract up() → rads.xml.base.Element
Get the parent of this element.

Returns Parent XML element.

Raises If there is no parent element.

abstract down() → rads.xml.base.Element
Get the first child of this element.

Returns First child XML element.

Raises **StopIteration** – If this element does not have any children.

property file
Get the name of the XML file containing this element.

Returns Name of the file containing this element, or None.

property opening_line
Get the opening line of the XML element.

Returns Opening line number, or None.

property num_lines
Get the number of lines making up the XML element.

Returns Number of lines in XML element, or None.

property closing_line
Get the closing line of the XML element.

Returns Closing line number, or None.

abstract property tag
Tag name of the element.

abstract property text
Internal text of the element.

abstract property attributes
The attributes of the element, as a dictionary.

rads.xml.etree module

XML tools using xml.etree.ElementTree.

exception rads.xml.etree.**ParseError**
Bases: SyntaxError

class rads.xml.etree.**XMLParser**
Bases: object

doctype()

feed()

close()

rads.xml.etree.**fromstring**(*text*, *parser=None*)
Parse XML document from string constant.

This function can be used to embed "XML Literals" in Python code.

text is a string containing XML data, *parser* is an optional parser instance, defaulting to the standard XMLParser.

Returns an Element instance.

rads.xml.etree.**fromstringlist**(*sequence*, *parser=None*)
Parse XML document from sequence of string fragments.

sequence is a list of other sequence, *parser* is an optional parser instance, defaulting to the standard XMLParser.

Returns an Element instance.

rads.xml.etree.**parse**(*source*, *parser=None*)
Parse XML document into element tree.

source is a filename or file object containing XML data, *parser* is an optional parser instance defaulting to XMLParser.

Return an ElementTree instance.

class rads.xml.etree.**Element**(*element: xml.etree.ElementTree.Element*, ***, *index: Optional[int] = None*, *parent: Optional[Element] = None*, *file: Optional[str] = None*)
Bases: *rads.xml.base.Element*

XML element that encapsulates an element from the ElementTree module.

Does not support line number examination.

Note: It is recommended to use *rads.xml.lxml.Element* if libxml is available on your system as the etree version does not support line numbers which can make debugging XML files for syntax errors more difficult.

Parameters

- **element** – XML element from the standard xml.etree.ElementTree package.
- **index** – Index of element at current level, among it's siblings. Not required if this element does not have any siblings.
- **parent** – The parent of this element.
- **file** – Filename of the XML document.

next() → rads.xml.etree.Element
Get the next sibling element.

Returns Next XML sibling element.

Raises **StopIteration** – If there is no next sibling element.

prev() → rads.xml.etree.Element
Get the previous sibling element.

Returns Previous XML sibling element.

Raises **StopIteration** – If there is no previous sibling element.

up() → rads.xml.etree.Element
Get the parent of this element.

Returns Parent XML element.

Raises If there is no parent element.

down() → rads.xml.etree.Element
Get the first child of this element.

Returns First child XML element.

Raises **StopIteration** – If this element does not have any children.

property file
Get the name of the XML file containing this element.

Returns Name of the file containing this element, or None.

property tag
Tag name of the element.

property text
Internal text of the element.

property attributes
The attributes of the element, as a dictionary.

rads.xml.etree.**error_with_file**(*error: xml.etree.ElementTree.ParseError*, *file: str*) → xml.etree.ElementTree.ParseError
Add filename to an XML parse error.

Parameters

- **error** – Original XML parse error.
- **file** – Filename to add.

Returns A new parse error (of the same type as *error*) with the *filename* added.

rads.xml.lxml module

XML tools using the lxml library.

rads.xml.lxml.**ParseError**

rads.xml.lxml.**XMLParser**

class rads.xml.lxml.**Element**(*element: lxml.etree._Element*, *, *file: Optional[str] = None*)
Bases: *rads.xml.base.Element*

XML element that encapsulates an element from lxml.

Supports line number examination.

Param XML element from the lxml library.

Parameters **file** – Optional filename/path the element is from.

next() → rads.xml.lxml.Element
Get the next sibling element.

Returns Next XML sibling element.

Raises **StopIteration** – If there is no next sibling element.

prev() → rads.xml.lxml.Element
Get the previous sibling element.

Returns Previous XML sibling element.

Raises **StopIteration** – If there is no previous sibling element.

up() → rads.xml.lxml.Element
Get the parent of this element.

Returns Parent XML element.

Raises If there is no parent element.

down() → rads.xml.lxml.Element
Get the first child of this element.

Returns First child XML element.

Raises **StopIteration** – If this element does not have any children.

property file
Get the name of the XML file containing this element.

Returns Name of the file containing this element, or None.

property opening_line
Get the opening line of the XML element.

Returns Opening line number, or None.

num_lines() → int
Get the number of lines making up the XML element.

Returns Number of lines in XML element, or None.

closing_line() → int
Get the closing line of the XML element.

Returns Closing line number, or None.

property tag
Tag name of the element.

property text
Internal text of the element.

property attributes
The attributes of the element, as a dictionary.

rads.xml.lxml.**parse**(*source: Union[str, bytes, int, IO[Any]]*, *parser: Optional[lxml.etree.XMLParser] = None*) → lxml.etree._ElementTree
Parse XML document into element tree.

This is wrapper around `lxml.etree.parse()` to make it behave like `xml.etree.ElementTree.parse()`.

Parameters

- **source** – Filename or file object containing XML data.
- **parser** – Optional parser instance, defaulting to `lxml.etree.ETCompatXMLParser`.

Returns An ElementTree instance.

rads.xml.lxml.**fromstring**(*text: Union[bytes, str], parser: Optional[lxml.etree.XMLParser] = None*) → lxml.etree._Element

Parse XML document from string constant.

This function can be used to embed 'XML Literals' in Python code.

This is wrapper around `lxml.etree.fromstring()` to make it behave like `xml.etree.ElementTree.fromtstring()`.

Parameters

- **text** – A string containing XML data.
- **parser** – Optional parser instance, defaulting to `lxml.etree.ETCompatXMLParser`.

Returns An Element instance.

rads.xml.lxml.**fromstringlist**(*sequence: Sequence[Union[bytes, str]], parser: Optional[lxml.etree.XMLParser] = None*) → lxml.etree._Element

Parse XML document from sequence of string fragments.

Parameters

- **sequence** – A list or other sequence of strings containing XML data.
- **parser** – Optional parser instance, defaulting to `lxml.etree.ETCompatXMLParser`.

Returns An Element instance.

rads.xml.lxml.**error_with_file**(*error: lxml.etree.ParseError*, *file: str*) → lxml.etree.ParseError

Add filename to an XML parse error.

Parameters

- **error** – Original XML parse error.
- **file** – Filename to add.

Returns A new parse error (of the same type as *error*) with the *filename* added.

rads.xml.utility module

XML tools for reading the RADS's configuration files.

rads.xml.utility.**parse**(*source: Union[str, os.PathLike, IO[Any]], parser: Optional[lxml.etree.XMLParser] = None, fixer: Optional[Callable[[str], str]] = None*) → rads.xml.lxml.Element

Parse an XML document from a file or file-like object.

Parameters

- **source** – File or file-like object containing the XML data.
- **parser** – XML parser to use, defaults to the standard XMLParser, which is ElementTree compatible regardless of backend.
- **fixer** – A function to pre-process the XML string. This can be used to fix files during load.

Returns The root XML element. If *rootless* is True this will be the added <*rootless*> element

rads.xml.utility.**fromstring**(*text: str, *, parser: Optional[lxml.etree.XMLParser] = None, fixer: Optional[Callable[[str], str]] = None, file: Optional[str] = None*) → rads.xml.lxml.Element

Parse an XML document or section from a string constant.

Parameters

- **text** – XML text to parse.
- **parser** – XML parser to use, defaults to the standard XMLParser, which is ElementTree compatible regardless of backend.
- **fixer** – A function to pre-process the XML string. This can be used to fix files during load.
- **file** – Optional filename to associate with the returned `xml.Element`.

Returns The root XML element (of the section given in *text*). If *rootless* is True this will be the added *<rootless>* element.

`rads.xml.utility.`**`fromstringlist`**(*sequence: Sequence[str], parser: Optional[lxml.etree.XMLParser] = None, fixer: Optional[Callable[[str], str]] = None, file: Optional[str] = None*) → rads.xml.lxml.Element

Parse an XML document or section from a sequence of string fragments.

Parameters

- **sequence** – String fragments containing the XML text to parse.
- **parser** – XML parser to use, defaults to the standard XMLParser, which is ElementTree compatible regardless of backend.
- **fixer** – A function to pre-process the XML string. This can be used to fix files during load. This will not be a string list but the full string with newlines.
- **file** – Optional filename to associate with the returned `xml.Element`.

Returns The root XML element (of the section given in *text*). If *rootless* is True this will be the added *<rootless>* element.

`rads.xml.utility.`**`rads_fixer`**(*text: str*) → str

Fix XML problems with the upstream RADS XML configuration.

This fixer is for problems that will not be fixed upstream or for which the fix has been delayed. It is primary for making up the difference between the official RADS parser which is very lenient and the PyRADS parser which is very strict.

Currently, this fixes the following bugs with the RADS config.

- The RADS XML file does not have a root as dictated by the XML 1.0 standard. This is fixed by adding <__ROOTLESS__> tags around the entire file. This is the only fix that is considered part of the RADS standard (that is RADS lies about it being XML 1.0).
- The RADS MXL file has some instances of *int3* used in the <compress> tag. This is an invalid type (there is no 3 byte integer) and in the official RADS implementation all invalid types default to *dble* (double). However, The intended type here is *int4*. This fix corrects this.

Parameters **text** – RADS XML string to fix.

Returns Repaired RADS XML string.

`rads.xml.utility.`**`rootless_fixer`**(*text: str, preserve_empty: bool = False*) → str

Fix rootless XML files.

Give this as the *fixer* argument in *parse()*, *fromstring()*, or *fromstringlist()* to load XML files that do not have a root tag. This is done by adding a <__ROOTLESS__> block around the entire document.

Parameters

- **text** – XML text to wrap <__ROOTLESS__> tags around.

- **preserve_empty** – Set to False to skip adding <__ROOTLESS__> tags to an empty XML file. See *is_empty()* for the definition of *empty*. In order to set this functools.partial() should be used.

Returns The given *text* with <__ROOTLESS__> tags added (after beginning processing instructions).

rads.xml.utility.**is_empty**(*text: str*) → bool
Determine if XML string is empty.

The XML string is considered empty if it only contains processing instructions and comments.

Parameters **text** – XML text to check for being empty.

Returns True if the given XML *text* is empty.

rads.xml.utility.**strip_comments**(*text: str*) → str
Remove XML comments from a string.

Note: This will not remove lines that had comments, it only removes the text from "<!–" to "–>".

Parameters **text** – XML text to strip comments from.

Returns The given *text* without XML comments.

rads.xml.utility.**strip_processing_instructions**(*text: str*) → str
Remove XML processing instructions from a string.

Note: This will not remove lines that had processing instructions, it only removes the text from "<?" to "?>".

Parameters **text** – XML text to strip processing instructions from.

Returns The given *text* without XML processing instructions.

rads.xml.utility.**strip_blanklines**(*text: str*) → str
Remove blank lines from a string.

Lines containing only whitespace characters are considered blank.

Parameters **text** – String to remove blank lines from.

Returns String without blank lines.

Module contents

XML library, specifically for reading the RADS's configuration files.

This includes *Element* which allows easy traversal of the XML tree new versions of the *parse()*, *fromstring()* and *fromstringlist()* functions for parsing and XML document that return *Element*. These functions also support XML documents without a root element, such as the RADS v4 configuration file.

class rads.xml.**Element**(*element: lxml.etree._Element*, *, *file: Optional[str] = None*)
Bases: *rads.xml.base.Element*

XML element that encapsulates an element from lxml.

Supports line number examination.

Param XML element from the lxml library.

Parameters **file** – Optional filename/path the element is from.

next() → rads.xml.lxml.Element
Get the next sibling element.

Returns Next XML sibling element.

Raises **StopIteration** – If there is no next sibling element.

prev() → rads.xml.lxml.Element
Get the previous sibling element.

Returns Previous XML sibling element.

Raises **StopIteration** – If there is no previous sibling element.

up() → rads.xml.lxml.Element
Get the parent of this element.

Returns Parent XML element.

Raises If there is no parent element.

down() → rads.xml.lxml.Element
Get the first child of this element.

Returns First child XML element.

Raises **StopIteration** – If this element does not have any children.

property file
Get the name of the XML file containing this element.

Returns Name of the file containing this element, or None.

property opening_line
Get the opening line of the XML element.

Returns Opening line number, or None.

num_lines() → int
Get the number of lines making up the XML element.

Returns Number of lines in XML element, or None.

closing_line() → int
Get the closing line of the XML element.

Returns Closing line number, or None.

property tag
Tag name of the element.

property text
Internal text of the element.

property attributes
The attributes of the element, as a dictionary.

rads.xml.**ParseError**

rads.xml.**fromstring**(*text: str, *, parser: Optional[lxml.etree.XMLParser] = None, fixer: Optional[Callable[[str], str]] = None, file: Optional[str] = None*) → rads.xml.lxml.Element
Parse an XML document or section from a string constant.

Parameters

- **text** – XML text to parse.
- **parser** – XML parser to use, defaults to the standard XMLParser, which is ElementTree compatible regardless of backend.
- **fixer** – A function to pre-process the XML string. This can be used to fix files during load.
- **file** – Optional filename to associate with the returned `xml.Element`.

Returns The root XML element (of the section given in *text*). If *rootless* is True this will be the added <*rootless*> element.

`rads.xml.`**`fromstringlist`**(*sequence: Sequence[str], parser: Optional[lxml.etree.XMLParser] = None, fixer: Optional[Callable[[str], str]] = None, file: Optional[str] = None*) → rads.xml.lxml.Element

Parse an XML document or section from a sequence of string fragments.

Parameters

- **sequence** – String fragments containing the XML text to parse.
- **parser** – XML parser to use, defaults to the standard XMLParser, which is ElementTree compatible regardless of backend.
- **fixer** – A function to pre-process the XML string. This can be used to fix files during load. This will not be a string list but the full string with newlines.
- **file** – Optional filename to associate with the returned `xml.Element`.

Returns The root XML element (of the section given in *text*). If *rootless* is True this will be the added <*rootless*> element.

`rads.xml.`**`parse`**(*source: Union[str, os.PathLike, IO[Any]], parser: Optional[lxml.etree.XMLParser] = None, fixer: Optional[Callable[[str], str]] = None*) → rads.xml.lxml.Element

Parse an XML document from a file or file-like object.

Parameters

- **source** – File or file-like object containing the XML data.
- **parser** – XML parser to use, defaults to the standard XMLParser, which is ElementTree compatible regardless of backend.
- **fixer** – A function to pre-process the XML string. This can be used to fix files during load.

Returns The root XML element. If *rootless* is True this will be the added <*rootless*> element

`rads.xml.`**`rads_fixer`**(*text: str*) → str

Fix XML problems with the upstream RADS XML configuration.

This fixer is for problems that will not be fixed upstream or for which the fix has been delayed. It is primary for making up the difference between the official RADS parser which is very lenient and the PyRADS parser which is very strict.

Currently, this fixes the following bugs with the RADS config.

- The RADS XML file does not have a root as dictated by the XML 1.0 standard. This is fixed by adding <__ROOTLESS__> tags around the entire file. This is the only fix that is considered part of the RADS standard (that is RADS lies about it being XML 1.0).
- The RADS MXL file has some instances of *int3* used in the <compress> tag. This is an invalid type (there is no 3 byte integer) and in the official RADS implementation all invalid types default to *dble* (double). However, The intended type here is *int4*. This fix corrects this.

Parameters **text** – RADS XML string to fix.

Returns Repaired RADS XML string.

rads.xml.**rootless_fixer**(*text: str*, *preserve_empty: bool = False*) → str
Fix rootless XML files.

Give this as the *fixer* argument in *parse()*, *fromstring()*, or *fromstringlist()* to load XML files that do not have a root tag. This is done by adding a <__ROOTLESS__> block around the entire document.

Parameters

- **text** – XML text to wrap <__ROOTLESS__> tags around.
- **preserve_empty** – Set to False to skip adding <__ROOTLESS__> tags to an empty XML file. See `is_empty()` for the definition of *empty*. In order to set this `functools.partial()` should be used.

Returns The given *text* with <__ROOTLESS__> tags added (after beginning processing instructions).

Submodules

rads.__version__ module

Project information, specifically the version.

rads.constants module

Constants for all of PyRADS.

rads.constants.**EPOCH = datetime.datetime(1985, 1, 1, 0, 0)**
RADS epoch, 1985-01-01 00:00:00 UTC

rads.datetime64util module

Additional utility for numpy.datetime64.

rads.datetime64util.**year**(*datetime64: numpy.datetime64*) → numpy.generic
Get year from NumPy datetime64 value/array.

Parameters **datetime64** – Value/array to get year number(s) from.

Returns Year or array of years from *datetime64*.

rads.datetime64util.**month**(*datetime64: numpy.datetime64*) → numpy.generic
Get month from NumPy datetime64 value/array.

Parameters **datetime64** – Value/array to get month number(s) from.

Returns Month or array of months from *datetime64*.

rads.datetime64util.**day**(*datetime64: numpy.datetime64*) → numpy.generic
Get day of month from NumPy datetime64 value/array.

Parameters **datetime64** – Value/array to get day(s) of month from.

Returns Day of month or array of days of month from *datetime64*.

rads.datetime64util.**hour**(*datetime64: numpy.datetime64*) → numpy.generic
Get hour from NumPy datetime64 value/array.

Parameters **datetime64** – Value/array to get hour(s) from.

Returns Hour or array of hours from *datetime64*.

rads.datetime64util.**minute**(*datetime64: numpy.datetime64*) → numpy.generic
Get minute from NumPy datetime64 value/array.

Parameters **datetime64** – Value/array to get minute(s) from.

Returns Minute or array of minutes from *datetime64*.

rads.datetime64util.**second**(*datetime64: numpy.datetime64*) → numpy.generic
Get second from NumPy datetime64 value/array.

Parameters **datetime64** – Value/array to get second(s) from.

Returns Second or array of seconds from *datetime64*.

rads.datetime64util.**microsecond**(*datetime64: numpy.datetime64*) → numpy.generic
Get microsecond from NumPy datetime64 value/array.

Parameters **datetime64** – Value/array to get microsecond(s) from.

Returns Microsecond or array of microseconds from *datetime64*.

rads.datetime64util.**ymdhmsus**(*datetime64: numpy.datetime64*) → Tuple[numpy.generic, numpy.generic, numpy.generic, numpy.generic, numpy.generic, numpy.generic, numpy.generic]
Get time components from NumPy datetime64 value/array.

Parameters **datetime64** – Value/array to get time components from.

Returns

A tuple with the following:

- Year or array of years from *datetime64*.
- Month or array of months from *datetime64*.
- Day of month or array of days of month from *datetime64*.
- Hour or array of hours from *datetime64*.
- Minute or array of minutes from *datetime64*.
- Second or array of seconds from *datetime64*.
- Microsecond or array of microseconds from *datetime64*.

rads.exceptions module

Public exceptions.

exception rads.exceptions.**RADSError**
Bases: Exception

Base class for all public PyRADS exceptions.

exception rads.exceptions.**ConfigError**(*message: str*, *line: Optional[int] = None*, *file: Optional[str] = None*, ***, *original: Optional[Exception] = None*)
Bases: *rads.exceptions.RADSError*

Exception raised when there is a problem loading the configuration file.

It is usually raised after another more specific exception has been caught.

Parameters

- **message** – Error message.
- **line** – Line that cause the exception, if known.
- **file** – File that caused the exception, if known.
- **original** – Optionally the original exception.

message
Error message.

line = None
Line that cause the exception, if known (None otherwise).

file = None
File that caused the exception, if known (None otherwise).

original_exception = None
Optionally the original exception (None otherwise).

exception rads.exceptions.**InvalidDataroot**
Bases: *rads.exceptions.RADSError*

Raised when the RADS dataroot is missing or invalid.

rads.rpn module

Reverse Polish Notation calculator.

Exceptions

- *StackUnderflowError*

Classes

- *Expression*
- *Token*
- *Literal*
- *Variable*
- *Operator*

Functions

- *token()*

Constants

Keyword	Value
PI	3.141592653589793
E	2.718281828459045

Operators

Keyword	Description
SUB	a = x - y
ADD	a = x + y
MUL	a = x*y
POP	remove top of stack
NEG	a = -x
ABS	a = |x|
INV	a = 1/x
SQRT	a = sqrt(x)
SQR	a = x*x
EXP	a = exp(x)
LOG	a = ln(x)
LOG10	a = log10(x)
SIN	a = sin(x)
COS	a = cos(x)
TAN	a = tan(x)
SIND	a = sin(x) [x in degrees]
COSD	a = cos(x) [x in degrees]
TAND	a = tan(x) [x in degrees]
SINH	a = sinh(x)
COSH	a = cosh(x)
TANH	a = tanh(x)
ASIN	a = arcsin(x)
ACOS	a = arccos(x)
ATAN	a = arctan(x)
ASIND	a = arcsin(x) [a in degrees]
ACOSD	a = arccos(x) [a in degrees]
ATAND	a = arctan(x) [a in degrees]
ASINH	a = arcsinh(x)
ACOSH	a = arccosh(x)
ATANH	a = arctanh(x)
ISNAN	a = 1 if x is NaN; a = 0 otherwise
ISAN	a = 0 if x is NaN; a = 1 otherwise
RINT	a is nearest integer to x
NINT	a is nearest integer to x
CEIL	a is nearest integer greater or equal to x
CEILING	a is nearest integer greater or equal to x
FLOOR	a is nearest integer less or equal to x
D2R	convert x from degrees to radians
R2D	convert x from radian to degrees

Continued on next page

Table 1 – continued from previous page

Keyword	Description
YMDHMS	convert from seconds since 1985 to YYMMDDHHMMSS format (float)
SUM	a[i] = x[1] + … + x[i] while skipping all NaN
DIF	a[i] = x[i]-x[i-1]; a[1] = NaN
DUP	duplicate the last item on the stack
DIV	a = x/y
POW	a = x**y
FMOD	a = x modulo y
MIN	a = the lesser of x and y [element wise]
MAX	a = the greater of x and y [element wise]
ATAN2	a = arctan2(x, y)
HYPOT	a = sqrt(x*x+y*y)
R2	a = x*x + y*y
EQ	a = 1 if x == y; a = 0 otherwise
NE	a = 0 if x == y; a = 1 otherwise
LT	a = 1 if x < y; a = 0 otherwise
LE	a = 1 if x ≤ y; a = 0 otherwise
GT	a = 1 if x > y; a = 0 otherwise
GE	a = 1 if x ≥ y; a = 0 otherwise
NAN	a = NaN if x == y; a = x otherwise
AND	a = y if x is NaN; a = x otherwise
OR	a = NaN if y is NaN; a = x otherwise
IAND	a = bitwise AND of x and y
IOR	a = bitwise OR of x and y
BTEST	a = 1 if bit y of x is set; a = 0 otherwise
AVG	a = 0.5*(x+y) [when x or y is NaN a returns the other value]
DXDY	a[i] = (x[i+1]-x[i-1])/(y[i+1]-y[i-1]); a[1] = a[n] = NaN
EXCH	exchange the top two stack elements
INRANGE	a = 1 if x is between y and z (inclusive); a = 0 otherwise
BOXCAR	filter x along dimension y with boxcar of length z
GAUSS	filter x along dimension y with Gaussian of width sigma z

exception rads.rpn.**StackUnderflowError**

Bases: Exception

Raised when the stack is too small for the operation.

When this is raised the stack will exist in the state that it was before the operation was attempted. Therefore, it is not necessary to repair the stack.

class rads.rpn.**Token**

Bases: abc.ABC

Base class of all RPN tokens.

See also:

Literal A literal numeric/array value.

Variable A variable to be looked up from the environment.

Operator Base class of operators that modify the stack.

abstract property pops

Elements removed off the stack by calling the token.

abstract property puts
Elements placed on the stack by calling the token.

abstract __call__(*stack: MutableSequence[Union[int, float, bool, numpy.generic, numpy.ndarray]], environment: Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]*) → None
Perform token's action on the given *stack*.

The actions currently supported are:

- Place literal value on the stack.
- Place variable on the stack from the *environment*.
- Perform operation on the stack.
- Any combination of the above.

Note: This must be overridden for all tokens.

Parameters

- **stack** – The stack of numbers/arrays to operate on.
- **environment** – The dictionary like object providing the immutable environment.

class rads.rpn.**Literal**(*value: Union[int, float, bool]*)
Bases: *rads.rpn.Token*

Literal value token.

Parameters **value** – Value of the literal.

Raises **ValueError** – If *value* is not a number.

property pops
Elements removed off the stack by calling the token.

property puts
Elements placed on the stack by calling the token.

property value
Value of the literal.

__call__(*stack: MutableSequence[Union[int, float, bool, numpy.generic, numpy.ndarray]], environment: Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]*) → None
Place literal value on top of the given *stack*.

Parameters

- **stack** – The stack of numbers/arrays to place the value on.
- **environment** – The dictionary like object providing the immutable environment. Not used by this method.

class rads.rpn.**Variable**(*name: str*)
Bases: *rads.rpn.Token*

Environment variable token.

This is a place holder to lookup and place a number/array from an environment mapping onto the stack.

Parameters **name** – Name of the variable, this is what will be used to lookup the variables value in the environment mapping.

property pops
Elements removed off the stack by calling the token.

property puts
Elements placed on the stack by calling the token.

property name
Name of the variable, used to lookup value in the environment.

__call__(*stack: MutableSequence[Union[int, float, bool, numpy.generic, numpy.ndarray]], environment: Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]*) → None
Get variable value from *environment* and place on stack.

Parameters

- **stack** – The stack of numbers/arrays to place value on.
- **environment** – The dictionary like object to lookup the variable's value from.

Raises **KeyError** – If the variable cannot be found in the given *environment*.

class rads.rpn.**Operator**(*name: str*)
Bases: *rads.rpn.Token*, abc.ABC

Base class of all RPN operators.

Parameters **name** – Name of the operator.

class rads.rpn.**Expression**(*tokens: Union[str, Iterable[Union[int, float, bool, str, rads.rpn.Token]]]*)
Bases: collections.abc.Sequence, typing.Generic, *rads.rpn.Token*

Reverse Polish Notation expression.

Note: *Expression*s cannot be evaluated as they may not be syntactically correct. For evaluation *CompleteExpression*s are required.

Expressions can be used in three ways:

- Can be converted to a *CompleteExpression* if *pops* and *puts* are both 1 with the *complete()* method.
- Can be added to the end of a *CompleteExpression* producing a *CompleteExpression* if the *Expression* has *pops* = 1 and *puts* = 1 or a *Expression* otherwise.
- Can be added to the end of an *Expression* producing a *CompleteExpression* if the combination produces an expression with *pops* = 0 and *puts* = 1.
- Can be used as a *Token* in another expression.

See also:

CompleteExpression For a expression that can be evaluated on it's own.

Parameters **tokens** – A Reverse Polish Notation expression given as a sequence of tokens or a string of tokens.

Note: This parameter is very forgiving. If given a sequence of tokens and some of the elements are not *Token*s then an attempt will be made to convert them to *Token*s. Because of this both numbers and strings can be given in the sequence of *tokens*.

---

pops() → int
Elements removed off the stack by calling the token.

puts() → int
Elements placed on the stack by calling the token.

variables() → AbstractSet[str]
Set of variables needed to evaluate the expression.

complete() → rads.rpn.CompleteExpression
Upgrade to a *CompleteExpression* if possible.

Returns A complete expression, assuming this partial expression takes zero inputs and provides one output.

Raises **ValueError** – If the partial expression is not a valid expression.

is_complete() → bool
Determine if can be upgraded to *CompleteExpression*.

Returns True if this expression can be upgraded to a *CompleteExpression* without error with the *complete()* method.

__call__(*stack: MutableSequence[Union[int, float, bool, numpy.generic, numpy.ndarray]], environment: Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]*) → None
Evaluate the expression as a token on the given *stack*.

Parameters

- **stack** – The stack of numbers/arrays to operate on.
- **environment** – The dictionary like object providing the immutable environment.

Raises ***StackUnderflowError*** – If the expression underflows the stack.

__add__(*other: Any*) → rads.rpn.Expression

class rads.rpn.**CompleteExpression**(*tokens: Union[str, Iterable[Union[int, float, bool, str, rads.rpn.Token]]]*)
Bases: *rads.rpn.Expression*

Reverse Polish Notation expression that can be evaluated.

Parameters **tokens** – A Reverse Polish Notation expression given as a sequence of tokens or a string of tokens.

Note: This parameter is very forgiving. If given a sequence of tokens and some of the elements are not *Token*s then then an attempt will be made to convert them to *Token*s. Because of this both numbers and strings can be given in the sequence of *tokens*.

Raises **ValueError** – If the sequence or string of *tokens* represents an invalid expression. This exception also indicates which token makes the expression invalid.

complete() → rads.rpn.CompleteExpression
Return this expression as it is already complete.

Returns This complete expression.

eval(*environment: Optional[Mapping[str, Union[int, float, bool, numpy.generic, numpy.ndarray]]] = None*) → Union[int, float, bool, numpy.generic, numpy.ndarray]
Evaluate the expression and return a numerical or logical result.

Parameters **environment** – A mapping to lookup variables in when evaluating the expression. If not provided an empty mapping will be used, this is fine as long as the expression does not contain any variables. This can be ascertained by checking the with the `variables` attribute:

```
if not expression.variables:
    expression.eval()
```

If the evaluation is lengthy or there are side effects to key lookup in the *environment* it may be beneficial to check for any missing variables first:

```
missing_vars = expression.variables.difference(environment)
```

Returns The numeric or logical result of the expression.

Raises

- **TypeError** – If there is a type mismatch with one of the operators and a value.

  Note: While this class includes a static syntax checker that runs upon initialization it does not know the type of variables in the given *environment* ahead of time.

- **KeyError** – If the expression contains a variable that is not within the given *environment*.
- **IndexError**, **ValueError**, **RuntimeError**, **ZeroDivisionError** – If arguments to operators in the expression do not have the proper dimensions or values for the operators to produce a result. See the documentation of each operator for specifics.

rads.rpn.**token**(*string: str*) → rads.rpn.Token

Parse string token into a *Token*.

There are three types of tokens that can result from this function:

- *Literal* - a literal integer or float
- *Variable* - a variable to looked up in the environment
- *Operator* - an operator to modify the stack

Parameters **string** – String to parse into a *Token*.

Returns Parsed token.

Raises

- **TypeError** – If not given a string.
- **ValueError** – If *string* is not a valid token.

rads.rpn.**SUB = SUB**

Subtract one number/array from another.

x y SUB a a = x - y

rads.rpn.**ADD = ADD**

Add two numbers/arrays.

x y ADD a a = x + y

rads.rpn.**MUL = MUL**

Multiply two numbers/arrays.

x y MUL a a = x*y

rads.rpn.**POP = POP**
Remove top of stack.

x POP remove last item from stack

rads.rpn.**NEG = NEG**
Negate number/array.

x NEG a a = -x

rads.rpn.**ABS = ABS**
Absolute value of number/array.

x ABS a a = |x|

rads.rpn.**INV = INV**
Invert number/array.

x INV a a = 1/x

rads.rpn.**SQRT = SQRT**
Compute square root of number/array.

x SQRT a a = sqrt(x)

rads.rpn.**SQR = SQR**
Square number/array.

x SQR a a = x*x

rads.rpn.**EXP = EXP**
Exponential of number/array.

x EXP a a = exp(x)

rads.rpn.**LOG = LOG**
Natural logarithm of number/array.

x LOG a a = ln(x)

rads.rpn.**LOG10 = LOG10**
Compute base 10 logarithm of number/array.

x LOG10 a a = log10(x)

rads.rpn.**SIN = SIN**
Sine of number/array [in radians].

x SIN a a = sin(x)

rads.rpn.**COS = COS**
Cosine of number/array [in radians].

x COS a a = cos(x)

rads.rpn.**TAN = TAN**
Tangent of number/array [in radians].

x TAN a a = tan(x)

rads.rpn.**SIND = SIND**
Sine of number/array [in degrees].

x SIND a a = sin(x) [x in degrees]

rads.rpn.**COSD = COSD**
Cosine of number/array [in degrees].

x COSD a a = cos(x) [x in degrees]

rads.rpn.**TAND = TAND**
Tangent of number/array [in degrees].

x TAND a a = tan(x) [x in degrees]

rads.rpn.**SINH = SINH**
Hyperbolic sine of number/array.

x SINH a a = sinh(x)

rads.rpn.**COSH = COSH**
Hyperbolic cosine of number/array.

x COSH a a = cosh(x)

rads.rpn.**TANH = TANH**
Hyperbolic tangent of number/array.

x TANH a a = tanh(x)

rads.rpn.**ASIN = ASIN**
Inverse sine of number/array [in radians].

x ASIN a a = arcsin(x)

rads.rpn.**ACOS = ACOS**
Inverse cosine of number/array [in radians].

x ACOS a a = arccos(x)

rads.rpn.**ATAN = ATAN**
Inverse tangent of number/array [in radians].

x ATAN a a = arctan(x)

rads.rpn.**ASIND = ASIND**
Inverse sine of number/array [in degrees].

x ASIND a a = arcsin(x) [a in degrees]

rads.rpn.**ACOSD = ACOSD**
Inverse cosine of number/array [in degrees].

x ACOSD a a = arccos(x) [a in degrees]

rads.rpn.**ATAND = ATAND**
Inverse tangent of number/array [in degrees].

x ATAND a a = arctan(x) [a in degrees]

rads.rpn.**ASINH = ASINH**
Inverse hyperbolic sine of number/array.

x ASINH a a = arcsinh(x)

rads.rpn.**ACOSH = ACOSH**
Inverse hyperbolic cosine of number/array.

x ACOSH a a = arccosh(x)

rads.rpn.**ATANH = ATANH**
Inverse hyperbolic tangent of number/array.

x ATANH a a = arctanh(x)

rads.rpn.**ISNAN = ISNAN**
Determine if number/array is NaN.

x ISNAN a a = 1 if x is NaN; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

rads.rpn.**ISAN = ISAN**
Determine if number/array is not NaN.

x ISAN a a = 0 if x is NaN; a = 1 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

rads.rpn.**RINT = RINT**
Round number/array to nearest integer.

x RINT a a is nearest integer to x

rads.rpn.**NINT = NINT**
Round number/array to nearest integer.

x NINT a a is nearest integer to x

rads.rpn.**CEIL = CEIL**
Round number/array up to nearest integer.

x CEIL a a is nearest integer greater or equal to x

rads.rpn.**CEILING = CEILING**
Round number/array up to nearest integer.

x CEILING a a is nearest integer greater or equal to x

rads.rpn.**FLOOR = FLOOR**
Round number/array down to nearest integer.

x FLOOR a a is nearest integer less or equal to x

rads.rpn.**D2R = D2R**
Convert number/array from degrees to radians.

x D2R a convert x from degrees to radians

rads.rpn.**R2D = R2D**
Convert number/array from radians to degrees.

x R2D a convert x from radian to degrees

rads.rpn.**YMDHMS = YMDHMS**
Convert number/array from seconds since RADS epoch to YYMMDDHHMMSS.

x YMDHMS a convert seconds of 1985 to format YYMMDDHHMMSS

Note: The top of the stack should be in seconds since the RADS epoch which is currently 1985-01-01 00:00:00 UTC

Note: The RADS documentation says this format uses a 4 digit year, but RADS uses a 2 digit year so that is what is used here.

rads.rpn.**SUM = SUM**
Compute sum over number/array [ignoring NaNs].

x SUM a a[i] = x[1] + … + x[i] while skipping all NaN

rads.rpn.**DIF = DIF**
Compute difference over number/array.

x DIF a a[i] = x[i]-x[i-1]; a[1] = NaN

rads.rpn.**DUP = DUP**
Duplicate top of stack.

x DUP a b duplicate the last item on the stack

Note: This is duplication by reference, no copy is made.

rads.rpn.**DIV = DIV**
Divide one number/array from another.

x y DIV a a = x/y

rads.rpn.**POW = POW**
Raise a number/array to the power of another number/array.

x y POW a a = x**y

rads.rpn.**FMOD = FMOD**
Remainder of dividing one number/array by another.

x y FMOD a a = x modulo y

rads.rpn.**MIN = MIN**
Minimum of two numbers/arrays [element wise].

x y MIN a a = the lesser of x and y

rads.rpn.**MAX = MAX**
Maximum of two numbers/arrays [element wise].

x y MAX a a = the greater of x and y

rads.rpn.**ATAN2 = ATAN2**
Inverse tangent of two numbers/arrays giving x and y.

x y ATAN2 a a = arctan2(x, y)

rads.rpn.**HYPOT = HYPOT**
Hypotenuse from numbers/arrays giving legs.

x y HYPOT a a = sqrt(x*x+y*y)

rads.rpn.**R2 = R2**
Sum of squares of two numbers/arrays.

x y R2 a a = x*x + y*y

rads.rpn.**EQ = EQ**
Compare two numbers/arrays for equality [element wise].

x y EQ a a = 1 if x == y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**NE = NE**
Compare two numbers/arrays for inequality [element wise].

x y NE a a = 0 if x == y; a = 1 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**LT = LT**
Compare two numbers/arrays with < [element wise].

x y LT a a = 1 if x < y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**LE = LE**
Compare two numbers/arrays with <= [element wise].

x y LE a a = 1 if x ≤ y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**GT = GT**
Compare two numbers/arrays with > [element wise].

x y GT a a = 1 if x > y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**GE = GE**
Compare two numbers/arrays with >= [element wise].

x y GE a a = 1 if x ≥ y; a = 0 otherwise

Note: Instead of using 1 and 0 pyrads uses True and False which behave as 1 and 0 when treated as numbers.

---

rads.rpn.**NAN = NAN**
Replace number/array with NaN where it is equal to another.

x y NAN a a = NaN if x == y; a = x otherwise

rads.rpn.**AND = AND**
Fallback to second number/array when first is NaN [element wise].

x y AND a a = y if x is NaN; a = x otherwise

rads.rpn.**OR = OR**
Replace number/array with NaN where second is NaN.

x y OR a a = NaN if y is NaN; a = x otherwise

rads.rpn.**IAND = IAND**
Bitwise AND of two numbers/arrays [element wise].

x y IAND a a = bitwise AND of x and y

rads.rpn.**IOR = IOR**
Bitwise OR of two numbers/arrays [element wise].

x y IOR a a = bitwise OR of x and y

rads.rpn.**BTEST = BTEST**
Test bit, given by second number/array, in first [element wise].

x y BTEST a a = 1 if bit y of x is set; a = 0 otherwise

rads.rpn.**AVG = AVG**
Average of two numbers/arrays ignoring NaNs [element wise].

x y AVG a a = 0.5*(x+y) [when x or y is NaN a returns the other value]

rads.rpn.**DXDY = DXDY**
Compute dx/dy from two numbers/arrays.

x y DXDY a a[i] = (x[i+1]-x[i-1])/(y[i+1]-y[i-1]); a[1] = a[n] = NaN

rads.rpn.**EXCH = EXCH**
Exchange top two elements of stack.

x y EXCH a b exchange the last two items on the stack (NaNs have no influence)

rads.rpn.**INRANGE = INRANGE**
Determine if number/array is between two numbers [element wise].

x y z INRANGE a a = 1 if x is between y and z (inclusive) a = 0 otherwise (also in case of any NaN)

rads.rpn.**BOXCAR = BOXCAR**
Filter number/array with a boxcar filter along a given dimension.

x y z BOXCAR a a = filter x along monotonic dimension y with boxcar of length z (NaNs are skipped)

Note: This may behave slightly differently than the official RADS software at boundaries and at NaN values.

Raises

- **IndexError** – If x does not have dimension y.
- **ValueError** – If y or z is not a scalar.

rads.rpn.**GAUSS = GAUSS**
Filter number/array with a gaussian filter along a given dimension.

x y z GAUSS a a = filter x along monotonic dimension y with Gauss function with sigma z (NaNs are skipped)

Raises

- **IndexError** – If x does not have dimension y.
- **ValueError** – If y or z is not a scalar.

rads.typing module

Type aliases.

rads.utility module

Utility functions.

rads.utility.**ensure_open**(*file: Union[str, os.PathLike, IO[Any]], mode: str = 'r', buffering: int = -1, encoding: Optional[str] = None, errors: Optional[str] = None, newline: Optional[str] = None, closefd: bool = True, closeio: bool = False*) → IO[Any]

Open file or leave file-like object open.

This function behaves identically to `open()` but can also accept a file-like object in the *file* parameter.

Parameters

- **file** – A path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped or a file-like object.

  Note: If a file descriptor is given, it is closed when the returned I/O object is closed, unless *closefd* is set to False.

  Note: If a file-like object is given closing the returned I/O object will not close the given file unless *closeio* is set to True.

- **mode** – See `open()`
- **buffering** – See `open()`
- **encoding** – See `open()`
- **errors** – See `open()`
- **newline** – See `open()`
- **closefd** – See `open()`
- **closeio** – If set to True then if *file* is a file like object it will be closed when either the __exit__ or close methods are called on the returned I/O object. By default these methods will be ignored when *file* is a file-like object.

Returns An I/O object or the original file-like object if *file* is a file-like object. If this is the original file-like object and *closeio* is set to False (the default) then it's close and __exit__ methods will be no-ops.

See also:

`open()`

rads.utility.**filestring**(*file: Union[str, os.PathLike, IO[Any]]*) → Optional[str]

Convert a PathLikeOrFile to a string.

Parameters **file** – file or file-like object to get the string for.

Returns The string representation of the filename or path. If it cannot get the name/path of the given file or file-like object or cannot convert it to a str, None will be returned.

rads.utility.**isio**(*obj: Any*, *, *read: bool = False*, *write: bool = False*) → bool
Determine if object is IO like and is read and/or write.

Note: Falls back to `isinstnace(obj, io.IOBase)` if neither *read* nor *write* is True.

Parameters

- **obj** – Object to check if it is an IO like object.
- **read** – Require *obj* to be readable if True.
- **write** – Require *obj* to be writable if True.

Returns True if the given *obj* is readable and/or writeable as defined by the *read* and *write* arguments.

rads.utility.**xor**(*a: bool*, *b: bool*) → bool
Boolean XOR operator.

This implements the XOR boolean operator and has the following truth table:

a	b	a XOR b
True	True	False
True	False	True
False	True	True
False	False	False

Parameters

- **a** – First boolean value.
- **b** – Second boolean value.

Returns The result of *a* XOR *b* from the truth table above.

rads.utility.**contains_sublist**(*list_: List[Any]*, *sublist: List[Any]*) → bool
Determine if a *list* contains a *sublist*.

Parameters

- **list** – list to search for the *sublist* in.
- **sublist** – Sub list to search for.

Returns True if *list* contains *sublist*.

rads.utility.**merge_sublist**(*list_: List[Any]*, *sublist: List[Any]*) → List[Any]
Merge a *sublist* into a given *list_*.

Parameters

- **list** – List to merge *sublist* into.
- **sublist** – Sublist to merge into *list_*

Returns A copy of *list_* with *sublist* at the end if *sublist* is not a sublist of *list_*. Otherwise, a copy of *list_* is returned unchanged.

rads.utility.**delete_sublist**(*list_: List[Any]*, *sublist: List[Any]*) → List[Any]
Remove a *sublist* from the given *list_*.

Parameters

- **list** – List to remove the *sublist* from.
- **sublist** – Sublist to remove from *list_*.

Returns A copy of *list_* with the *sublist* removed.

rads.utility.**fortran_float**(*string: str*) → float

Construct `float` from Fortran style float strings.

This function can convert strings to floats in all of the formats below:

- `3.14e10` (also parsable with `float`)
- `3.14E10` (also parsable with `float`)
- `3.14d10`
- `3.14D10`
- `3.14e+10` (also parsable with `float`)
- `3.14E+10` (also parsable with `float`)
- `3.14d+10`
- `3.14D+10`
- `3.14e-10` (also parsable with `float`)
- `3.14E-10` (also parsable with `float`)
- `3.14d-10`
- `3.14D-10`
- `3.14+100`
- `3.14-100`

Note: Because RADS was written in Fortran, exponent characters in configuration and passindex files sometimes use 'D' or 'd' as the exponent separator instead of 'E' or 'e'.

Warning: If you are Fortran developer stop using 'Ew.d' and 'Ew.dDe' formats and use 'Ew.dEe' instead. The first two are not commonly supported by other languages while the last version is the standard for nearly all languages. Ok, rant over.

Parameters **string** – String to attempt to convert to a float.

Returns The float parsed from the given *string*.

Raises **ValueError** – If *string* does not represent a valid float.

Module contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

rads.__version__, 79
rads.config, 61
rads.config.builders, 29
rads.config.text_parsers, 34
rads.config.tree, 39
rads.config.xml_parsers, 48
rads.constants, 79
rads.datetime64util, 79
rads.exceptions, 80
rads.rpn, 81
rads.typing, 94
rads.utility, 94
rads.xml, 76
rads.xml.base, 69
rads.xml.etree, 70
rads.xml.lxml, 72
rads.xml.utility, 74

INDEX

Symbols

__add__() (*rads.config.xml_parsers.Parser method*), 50
__add__() (*rads.config.xml_parsers.Sequence method*), 55
__add__() (*rads.rpn.Expression method*), 86
__call__() (*rads.config.xml_parsers.Alternate method*), 56
__call__() (*rads.config.xml_parsers.AnyElement method*), 58
__call__() (*rads.config.xml_parsers.Apply method*), 51
__call__() (*rads.config.xml_parsers.At method*), 53
__call__() (*rads.config.xml_parsers.End method*), 58
__call__() (*rads.config.xml_parsers.Failure method*), 57
__call__() (*rads.config.xml_parsers.Lazy method*), 52
__call__() (*rads.config.xml_parsers.Must method*), 53
__call__() (*rads.config.xml_parsers.Not method*), 54
__call__() (*rads.config.xml_parsers.Parser method*), 50
__call__() (*rads.config.xml_parsers.Repeat method*), 54
__call__() (*rads.config.xml_parsers.Sequence method*), 55
__call__() (*rads.config.xml_parsers.Start method*), 57
__call__() (*rads.config.xml_parsers.Success method*), 56
__call__() (*rads.config.xml_parsers.Tag method*), 59
__call__() (*rads.rpn.Expression method*), 86
__call__() (*rads.rpn.Literal method*), 84
__call__() (*rads.rpn.Token method*), 84
__call__() (*rads.rpn.Variable method*), 85
__iadd__() (*rads.config.xml_parsers.Sequence method*), 55
__invert__() (*rads.config.xml_parsers.Parser method*), 51
__lshift__() (*rads.config.xml_parsers.Parser method*), 51
__or__() (*rads.config.xml_parsers.Alternate method*), 56
__or__() (*rads.config.xml_parsers.Parser method*), 50
__xor__() (*rads.config.xml_parsers.Parser method*), 51

A

ABS (*in module rads.rpn*), 88
absolute_orbit_number (*rads.config.ReferencePass attribute*), 65
absolute_orbit_number (*rads.config.tree.ReferencePass attribute*), 40
ACOS (*in module rads.rpn*), 89
ACOSD (*in module rads.rpn*), 89
ACOSH (*in module rads.rpn*), 89
ADD (*in module rads.rpn*), 87
add_offset (*rads.config.Compress attribute*), 62
add_offset (*rads.config.tree.Compress attribute*), 41
aliases (*rads.config.Satellite attribute*), 66
aliases (*rads.config.tree.Satellite attribute*), 47
Alternate (*class in rads.config.xml_parsers*), 55
AND (*in module rads.rpn*), 92
any() (*in module rads.config.xml_parsers*), 61
AnyElement (*class in rads.config.xml_parsers*), 58
Apply (*class in rads.config.xml_parsers*), 51
ASIN (*in module rads.rpn*), 89
ASIND (*in module rads.rpn*), 89
ASINH (*in module rads.rpn*), 89
At (*class in rads.config.xml_parsers*), 53
at() (*in module rads.config.xml_parsers*), 59
ATAN (*in module rads.rpn*), 89
ATAN2 (*in module rads.rpn*), 91
ATAND (*in module rads.rpn*), 89
ATANH (*in module rads.rpn*), 89
attributes() (*rads.xml.base.Element property*), 70
attributes() (*rads.xml.Element property*), 77
attributes() (*rads.xml.etree.Element property*), 72
attributes() (*rads.xml.lxml.Element property*), 73
AVG (*in module rads.rpn*), 93

B

bit (*rads.config.MultiBitFlag attribute*), 63
bit (*rads.config.SingleBitFlag attribute*), 67
bit (*rads.config.tree.MultiBitFlag attribute*), 42
bit (*rads.config.tree.SingleBitFlag attribute*), 43
blacklist (*rads.config.tree.PreConfig attribute*), 39
BOXCAR (*in module rads.rpn*), 93

branch (*rads.config.NetCDFAttribute attribute*), 63
branch (*rads.config.NetCDFVariable attribute*), 64
branch (*rads.config.tree.NetCDFAttribute attribute*), 44
branch (*rads.config.tree.NetCDFVariable attribute*), 44
BTEST (*in module rads.rpn*), 93
build() (*rads.config.builders.PhaseBuilder method*), 31
build() (*rads.config.builders.PreConfigBuilder method*), 34
build() (*rads.config.builders.SatelliteBuilder method*), 30
build() (*rads.config.builders.VariableBuilder method*), 33

C

CEIL (*in module rads.rpn*), 90
CEILING (*in module rads.rpn*), 90
close() (*rads.xml.etree.XMLParser method*), 71
closing_line() (*rads.xml.base.Element property*), 70
closing_line() (*rads.xml.Element method*), 77
closing_line() (*rads.xml.lxml.Element method*), 73
comment (*rads.config.tree.Variable attribute*), 45
comment (*rads.config.Variable attribute*), 68
complete() (*rads.rpn.CompleteExpression method*), 86
complete() (*rads.rpn.Expression method*), 86
CompleteExpression (*class in rads.rpn*), 86
Compress (*class in rads.config*), 61
Compress (*class in rads.config.tree*), 41
compress (*rads.config.tree.Variable attribute*), 46
compress (*rads.config.Variable attribute*), 69
compress() (*in module rads.config.text_parsers*), 36
Config (*class in rads.config*), 62
Config (*class in rads.config.tree*), 47
config_files (*rads.config.Config attribute*), 62
config_files (*rads.config.tree.Config attribute*), 47
config_files (*rads.config.tree.PreConfig attribute*), 39
ConfigError, 80
Constant (*class in rads.config*), 62
Constant (*class in rads.config.tree*), 42
contains_sublist() (*in module rads.utility*), 95
COS (*in module rads.rpn*), 88
COSD (*in module rads.rpn*), 88
COSH (*in module rads.rpn*), 89
cycle_number (*rads.config.ReferencePass attribute*), 65
cycle_number (*rads.config.tree.ReferencePass attribute*), 40
Cycles (*class in rads.config*), 62
Cycles (*class in rads.config.tree*), 39
cycles (*rads.config.Phase attribute*), 64
cycles (*rads.config.tree.Phase attribute*), 41
cycles() (*in module rads.config.text_parsers*), 36

D

D2R (*in module rads.rpn*), 90
data (*rads.config.tree.Variable attribute*), 45
data (*rads.config.Variable attribute*), 68
data() (*in module rads.config.text_parsers*), 37
dataroot (*rads.config.Config attribute*), 62
dataroot (*rads.config.tree.Config attribute*), 47
dataroot (*rads.config.tree.PreConfig attribute*), 39
day() (*in module rads.datetime64util*), 79
days (*rads.config.Repeat attribute*), 65
days (*rads.config.tree.Repeat attribute*), 40
default (*rads.config.tree.Variable attribute*), 46
default (*rads.config.Variable attribute*), 69
delete_sublist() (*in module rads.utility*), 95
DIF (*in module rads.rpn*), 91
dimensions (*rads.config.tree.Variable attribute*), 46
dimensions (*rads.config.Variable attribute*), 69
DIV (*in module rads.rpn*), 91
doctype() (*rads.xml.etree.XMLParser method*), 71
down() (*rads.xml.base.Element method*), 70
down() (*rads.xml.Element method*), 77
down() (*rads.xml.etree.Element method*), 72
down() (*rads.xml.lxml.Element method*), 73
dt1hz (*rads.config.Satellite attribute*), 66
dt1hz (*rads.config.tree.Satellite attribute*), 46
dumps() (*rads.xml.base.Element method*), 69
DUP (*in module rads.rpn*), 91
DXDY (*in module rads.rpn*), 93

E

Element (*class in rads.xml*), 76
Element (*class in rads.xml.base*), 69
Element (*class in rads.xml.etree*), 71
Element (*class in rads.xml.lxml*), 72
End (*class in rads.config.xml_parsers*), 58
end() (*in module rads.config.xml_parsers*), 61
end_time (*rads.config.Phase attribute*), 64
end_time (*rads.config.tree.Phase attribute*), 41
ensure_open() (*in module rads.utility*), 94
EPOCH (*in module rads.constants*), 79
EQ (*in module rads.rpn*), 91
error_with_file() (*in module rads.xml.etree*), 72
error_with_file() (*in module rads.xml.lxml*), 74
eval() (*rads.rpn.CompleteExpression method*), 86
EXCH (*in module rads.rpn*), 93
EXP (*in module rads.rpn*), 88
Expression (*class in rads.rpn*), 85
extract() (*rads.config.MultiBitFlag method*), 63
extract() (*rads.config.SingleBitFlag method*), 67
extract() (*rads.config.SurfaceType method*), 67
extract() (*rads.config.tree.Flags method*), 42
extract() (*rads.config.tree.MultiBitFlag method*), 42
extract() (*rads.config.tree.SingleBitFlag method*), 43
extract() (*rads.config.tree.SurfaceType method*), 43

F

Failure (*class in rads.config.xml_parsers*), 57

failure() (*in module rads.config.xml_parsers*), 61
feed() (*rads.xml.etree.XMLParser method*), 71
fields() (*rads.config.builders.PhaseBuilder method*), 31
fields() (*rads.config.builders.PreConfigBuilder method*), 34
fields() (*rads.config.builders.SatelliteBuilder method*), 30
fields() (*rads.config.builders.VariableBuilder method*), 33
file (*rads.config.Grid attribute*), 63
file (*rads.config.tree.Grid attribute*), 43
file (*rads.config.xml_parsers.TerminalXMLParseError attribute*), 49
file (*rads.exceptions.ConfigError attribute*), 81
file() (*rads.xml.base.Element property*), 70
file() (*rads.xml.Element property*), 77
file() (*rads.xml.etree.Element property*), 72
file() (*rads.xml.lxml.Element property*), 73
filestring() (*in module rads.utility*), 94
first (*rads.config.Cycles attribute*), 62
first (*rads.config.tree.Cycles attribute*), 40
first_child() (*in module rads.config.xml_parsers*), 50
flag_masks (*rads.config.tree.Variable attribute*), 46
flag_masks (*rads.config.Variable attribute*), 69
flag_values (*rads.config.tree.Variable attribute*), 46
flag_values (*rads.config.Variable attribute*), 68
Flags (*class in rads.config.tree*), 42
FLOOR (*in module rads.rpn*), 90
FMOD (*in module rads.rpn*), 91
format (*rads.config.tree.Variable attribute*), 46
format (*rads.config.Variable attribute*), 69
fortran_float() (*in module rads.utility*), 96
frequency (*rads.config.Satellite attribute*), 66
frequency (*rads.config.tree.Satellite attribute*), 47
fromstring() (*in module rads.xml*), 77
fromstring() (*in module rads.xml.etree*), 71
fromstring() (*in module rads.xml.lxml*), 73
fromstring() (*in module rads.xml.utility*), 74
fromstringlist() (*in module rads.xml*), 78
fromstringlist() (*in module rads.xml.etree*), 71
fromstringlist() (*in module rads.xml.lxml*), 74
fromstringlist() (*in module rads.xml.utility*), 75
full_string() (*rads.config.Config method*), 62
full_string() (*rads.config.Satellite method*), 66
full_string() (*rads.config.tree.Config method*), 47
full_string() (*rads.config.tree.Satellite method*), 47

G

GAUSS (*in module rads.rpn*), 93
GE (*in module rads.rpn*), 92
Grid (*class in rads.config*), 62
Grid (*class in rads.config.tree*), 43
GT (*in module rads.rpn*), 92

H

hour() (*in module rads.datetime64util*), 79
HYPOT (*in module rads.rpn*), 91

I

IAND (*in module rads.rpn*), 92
id (*rads.config.Phase attribute*), 64
id (*rads.config.Satellite attribute*), 66
id (*rads.config.tree.Phase attribute*), 41
id (*rads.config.tree.Satellite attribute*), 46
id (*rads.config.tree.Variable attribute*), 45
id (*rads.config.Variable attribute*), 68
id3 (*rads.config.Satellite attribute*), 66
id3 (*rads.config.tree.Satellite attribute*), 46
inclination (*rads.config.Satellite attribute*), 66
inclination (*rads.config.tree.Satellite attribute*), 47
INRANGE (*in module rads.rpn*), 93
INV (*in module rads.rpn*), 88
InvalidDataroot, 81
IOR (*in module rads.rpn*), 93
is_complete() (*rads.rpn.Expression method*), 86
is_empty() (*in module rads.xml.utility*), 76
ISAN (*in module rads.rpn*), 90
isio() (*in module rads.utility*), 94
ISNAN (*in module rads.rpn*), 89

L

last (*rads.config.Cycles attribute*), 62
last (*rads.config.tree.Cycles attribute*), 40
Lazy (*class in rads.config.xml_parsers*), 52
lazy() (*in module rads.config.xml_parsers*), 59
LE (*in module rads.rpn*), 92
length (*rads.config.MultiBitFlag attribute*), 63
length (*rads.config.tree.MultiBitFlag attribute*), 42
lengths (*rads.config.SubCycles attribute*), 67
lengths (*rads.config.tree.SubCycles attribute*), 40
lift() (*in module rads.config.text_parsers*), 34
limits (*rads.config.tree.Variable attribute*), 46
limits (*rads.config.Variable attribute*), 69
line (*rads.config.xml_parsers.TerminalXMLParseError attribute*), 49
line (*rads.exceptions.ConfigError attribute*), 81
list_of() (*in module rads.config.text_parsers*), 35
Literal (*class in rads.rpn*), 84
LOG (*in module rads.rpn*), 88
LOG10 (*in module rads.rpn*), 88
longitude (*rads.config.ReferencePass attribute*), 65
longitude (*rads.config.tree.ReferencePass attribute*), 40
longitude_drift (*rads.config.Repeat attribute*), 65
longitude_drift (*rads.config.tree.Repeat attribute*), 40
LT (*in module rads.rpn*), 92

M

MAX (*in module rads.rpn*), 91
max (*rads.config.Range attribute*), 65
max (*rads.config.tree.Range attribute*), 44
merge_sublist() (*in module rads.utility*), 95
message (*rads.config.xml_parsers.TerminalXMLParseError attribute*), 49
message (*rads.exceptions.ConfigError attribute*), 81
method (*rads.config.Grid attribute*), 63
method (*rads.config.tree.Grid attribute*), 43
microsecond() (*in module rads.datetime64util*), 80
MIN (*in module rads.rpn*), 91
min (*rads.config.Range attribute*), 65
min (*rads.config.tree.Range attribute*), 44
minute() (*in module rads.datetime64util*), 80
mission (*rads.config.Phase attribute*), 64
mission (*rads.config.tree.Phase attribute*), 41
month() (*in module rads.datetime64util*), 79
MUL (*in module rads.rpn*), 87
MultiBitFlag (*class in rads.config*), 63
MultiBitFlag (*class in rads.config.tree*), 42
Must (*class in rads.config.xml_parsers*), 52
must() (*in module rads.config.xml_parsers*), 60

N

name (*rads.config.NetCDFAttribute attribute*), 63
name (*rads.config.NetCDFVariable attribute*), 64
name (*rads.config.Satellite attribute*), 66
name (*rads.config.tree.NetCDFAttribute attribute*), 44
name (*rads.config.tree.NetCDFVariable attribute*), 44
name (*rads.config.tree.Satellite attribute*), 46
name (*rads.config.tree.Variable attribute*), 45
name (*rads.config.Variable attribute*), 68
name() (*rads.rpn.Variable property*), 85
names (*rads.config.Satellite attribute*), 66
names (*rads.config.tree.Satellite attribute*), 46
NAN (*in module rads.rpn*), 92
NE (*in module rads.rpn*), 92
NEG (*in module rads.rpn*), 88
NetCDFAttribute (*class in rads.config*), 63
NetCDFAttribute (*class in rads.config.tree*), 44
NetCDFVariable (*class in rads.config*), 64
NetCDFVariable (*class in rads.config.tree*), 44
next() (*rads.xml.base.Element method*), 69
next() (*rads.xml.Element method*), 77
next() (*rads.xml.etree.Element method*), 71
next() (*rads.xml.lxml.Element method*), 72
next_element() (*in module rads.config.xml_parsers*), 50
NINT (*in module rads.rpn*), 90
nop() (*in module rads.config.text_parsers*), 38
Not (*class in rads.config.xml_parsers*), 53
not_at() (*in module rads.config.xml_parsers*), 59
num_lines() (*rads.xml.base.Element property*), 70
num_lines() (*rads.xml.Element method*), 77
num_lines() (*rads.xml.lxml.Element method*), 73

O

one_of() (*in module rads.config.text_parsers*), 35
opening_line() (*rads.xml.base.Element property*), 70
opening_line() (*rads.xml.Element property*), 77
opening_line() (*rads.xml.lxml.Element property*), 73
Operator (*class in rads.rpn*), 85
opt() (*in module rads.config.xml_parsers*), 59
OR (*in module rads.rpn*), 92
original_exception (*rads.exceptions.ConfigError attribute*), 81

P

parse() (*in module rads.xml*), 78
parse() (*in module rads.xml.etree*), 71
parse() (*in module rads.xml.lxml*), 73
parse() (*in module rads.xml.utility*), 74
ParseError, 70
ParseError (*in module rads.xml*), 77
ParseError (*in module rads.xml.lxml*), 72
Parser (*class in rads.config.xml_parsers*), 50
pass_number (*rads.config.ReferencePass attribute*), 65
pass_number (*rads.config.tree.ReferencePass attribute*), 40
passes (*rads.config.Repeat attribute*), 65
passes (*rads.config.tree.Repeat attribute*), 40
Phase (*class in rads.config*), 64
Phase (*class in rads.config.tree*), 40
PhaseBuilder (*class in rads.config.builders*), 30
phases (*rads.config.Satellite attribute*), 66
phases (*rads.config.tree.Satellite attribute*), 47
plot_range (*rads.config.tree.Variable attribute*), 46
plot_range (*rads.config.Variable attribute*), 69
plus() (*in module rads.config.xml_parsers*), 60
POP (*in module rads.rpn*), 87
pops() (*rads.rpn.Expression method*), 85
pops() (*rads.rpn.Literal property*), 84
pops() (*rads.rpn.Token property*), 83
pops() (*rads.rpn.Variable property*), 84
POW (*in module rads.rpn*), 91
PreConfig (*class in rads.config.tree*), 39
PreConfigBuilder (*class in rads.config.builders*), 33
prev() (*rads.xml.base.Element method*), 70
prev() (*rads.xml.Element method*), 77
prev() (*rads.xml.etree.Element method*), 71
prev() (*rads.xml.lxml.Element method*), 73
puts() (*rads.rpn.Expression method*), 86
puts() (*rads.rpn.Literal property*), 84
puts() (*rads.rpn.Token property*), 83
puts() (*rads.rpn.Variable property*), 85

Q

quality_flag (*rads.config.tree.Variable attribute*), 46
quality_flag (*rads.config.Variable attribute*), 69

R

R2 (*in module rads.rpn*), 91
R2D (*in module rads.rpn*), 90
rads.__version__ (*module*), 79
rads.config (*module*), 61
rads.config.builders (*module*), 29
rads.config.text_parsers (*module*), 34
rads.config.tree (*module*), 39
rads.config.xml_parsers (*module*), 48
rads.constants (*module*), 79
rads.datetime64util (*module*), 79
rads.exceptions (*module*), 80
rads.rpn (*module*), 81
rads.typing (*module*), 94
rads.utility (*module*), 94
rads.xml (*module*), 76
rads.xml.base (*module*), 69
rads.xml.etree (*module*), 70
rads.xml.lxml (*module*), 72
rads.xml.utility (*module*), 74
rads_fixer() (*in module rads.xml*), 78
rads_fixer() (*in module rads.xml.utility*), 75
RADSError, 80
Range (*class in rads.config*), 65
Range (*class in rads.config.tree*), 44
range_of() (*in module rads.config.text_parsers*), 35
ref_pass() (*in module rads.config.text_parsers*), 38
reference_pass (*rads.config.Phase attribute*), 64
reference_pass (*rads.config.tree.Phase attribute*), 41
ReferencePass (*class in rads.config*), 65
ReferencePass (*class in rads.config.tree*), 40
rep() (*in module rads.config.xml_parsers*), 60
Repeat (*class in rads.config*), 65
Repeat (*class in rads.config.tree*), 40
Repeat (*class in rads.config.xml_parsers*), 54
repeat (*rads.config.Phase attribute*), 64
repeat (*rads.config.tree.Phase attribute*), 41
repeat() (*in module rads.config.text_parsers*), 38
RINT (*in module rads.rpn*), 90
rootless_fixer() (*in module rads.xml*), 79
rootless_fixer() (*in module rads.xml.utility*), 75

S

Satellite (*class in rads.config*), 65
Satellite (*class in rads.config.tree*), 46
SatelliteBuilder (*class in rads.config.builders*), 29
satellites (*rads.config.Config attribute*), 62
satellites (*rads.config.tree.Config attribute*), 47
satellites (*rads.config.tree.PreConfig attribute*), 39
scale_factor (*rads.config.Compress attribute*), 62
scale_factor (*rads.config.tree.Compress attribute*), 41
second() (*in module rads.datetime64util*), 80
seq() (*in module rads.config.xml_parsers*), 60
Sequence (*class in rads.config.xml_parsers*), 54
SIN (*in module rads.rpn*), 88
SIND (*in module rads.rpn*), 88
SingleBitFlag (*class in rads.config*), 66
SingleBitFlag (*class in rads.config.tree*), 42
SINH (*in module rads.rpn*), 89
sor() (*in module rads.config.xml_parsers*), 60
source (*rads.config.tree.Variable attribute*), 45
source (*rads.config.Variable attribute*), 68
SQR (*in module rads.rpn*), 88
SQRT (*in module rads.rpn*), 88
StackUnderflowError, 83
standard_name (*rads.config.tree.Variable attribute*), 45
standard_name (*rads.config.Variable attribute*), 68
star() (*in module rads.config.xml_parsers*), 60
Start (*class in rads.config.xml_parsers*), 57
start (*rads.config.SubCycles attribute*), 67
start (*rads.config.tree.SubCycles attribute*), 40
start() (*in module rads.config.xml_parsers*), 61
start_time (*rads.config.Phase attribute*), 64
start_time (*rads.config.tree.Phase attribute*), 41
strip_blanklines() (*in module rads.xml.utility*), 76
strip_comments() (*in module rads.xml.utility*), 76
strip_processing_instructions() (*in module rads.xml.utility*), 76
SUB (*in module rads.rpn*), 87
SubCycles (*class in rads.config*), 67
SubCycles (*class in rads.config.tree*), 40
subcycles (*rads.config.Phase attribute*), 64
subcycles (*rads.config.tree.Phase attribute*), 41
Success (*class in rads.config.xml_parsers*), 56
success() (*in module rads.config.xml_parsers*), 61
SUM (*in module rads.rpn*), 91
SurfaceType (*class in rads.config*), 67
SurfaceType (*class in rads.config.tree*), 43

T

Tag (*class in rads.config.xml_parsers*), 59
tag() (*in module rads.config.xml_parsers*), 61
tag() (*rads.xml.base.Element property*), 70
tag() (*rads.xml.Element property*), 77
tag() (*rads.xml.etree.Element property*), 72
tag() (*rads.xml.lxml.Element property*), 73
TAN (*in module rads.rpn*), 88
TAND (*in module rads.rpn*), 89
TANH (*in module rads.rpn*), 89
terminal() (*rads.config.xml_parsers.XMLParseError method*), 50
TerminalTextParseError, 34
TerminalXMLParseError, 49

text() (*rads.xml.base.Element property*), 70
text() (*rads.xml.Element property*), 77
text() (*rads.xml.etree.Element property*), 72
text() (*rads.xml.lxml.Element property*), 73
TextParseError, 34
time (*rads.config.ReferencePass attribute*), 65
time (*rads.config.tree.ReferencePass attribute*), 40
time() (*in module rads.config.text_parsers*), 38
Token (*class in rads.rpn*), 83
token() (*in module rads.rpn*), 87
type (*rads.config.Compress attribute*), 62
type (*rads.config.tree.Compress attribute*), 41

U

unit() (*in module rads.config.text_parsers*), 39
units (*rads.config.tree.Variable attribute*), 45
units (*rads.config.Variable attribute*), 68
until() (*in module rads.config.xml_parsers*), 61
up() (*rads.xml.base.Element method*), 70
up() (*rads.xml.Element method*), 77
up() (*rads.xml.etree.Element method*), 72
up() (*rads.xml.lxml.Element method*), 73

V

value (*rads.config.Constant attribute*), 62
value (*rads.config.tree.Constant attribute*), 42
value() (*rads.rpn.Literal property*), 84
Variable (*class in rads.config*), 67
Variable (*class in rads.config.tree*), 44
Variable (*class in rads.rpn*), 84
variable (*rads.config.NetCDFAttribute attribute*), 63
variable (*rads.config.tree.NetCDFAttribute attribute*), 44
VariableBuilder (*class in rads.config.builders*), 31
variables (*rads.config.Satellite attribute*), 66
variables (*rads.config.tree.Satellite attribute*), 47
variables() (*rads.rpn.Expression method*), 86

X

x (*rads.config.Grid attribute*), 63
x (*rads.config.tree.Grid attribute*), 43
XMLParseError, 49
XMLParser (*class in rads.xml.etree*), 70
XMLParser (*in module rads.xml.lxml*), 72
xor() (*in module rads.utility*), 95

Y

y (*rads.config.Grid attribute*), 63
y (*rads.config.tree.Grid attribute*), 43
year() (*in module rads.datetime64util*), 79
YMDHMS (*in module rads.rpn*), 90
ymdhmsus() (*in module rads.datetime64util*), 80