understandAPI

```
This is the python interface to Understand databases.

It provides class-orientated access to Understand databases. Most
of the class objects are only valid when returned from a function.

The following classes and methods are in this module:
Classes:
  understand.Arch
  understand.Db
  understand.Ent
  understand.Kind
  understand.Lexeme
  understand.Lexer
  understand.LexerIter
  understand.Metric
  understand.Ref
  understand.UnderstandError
  understand.Visio
Methods:
  understand.checksum(text [,len])
  understand.license(path)
  understand.open(dbname)
  understand.version()

Examples

The following examples are meant to be complete, yet simplistic
scripts that demonstrate one or more features each. For the sake of
brevity, most try, except statements statements are ommitted.

Sorted List of All Entities
---------------------------

import understand

# Open Database
db = understand.open("test.udb")

for ent in sorted(db.ents(),key= lambda ent: ent.name()):
  print (ent.name(),"  [",ent.kindname(),"]",sep="",end="\n")


List of Files
-------------

import understand

db = understand.open("test.udb")

for file in db.ents("File"):
  # print directory name
  print (file.longname())


Lookup an Entity (Case Insensitive)
-----------------------------------

import understand
import re
```

```
db = understand.open("test.udb")

# Create a regular expression that is case insensitive
searchstr = re.compile("test*.cpp",re.I)
for file in db.lookup(searchstr,"File"):
  print (file)
```

```
Global Variable Usage
---------------------
```

```
import understand

db = understand.open("test.udb")

for ent in db.ents("Global Object ~Static"):
  print (ent,":",sep="")
  for ref in ent.refs():
    print (ref.kindname(),ref.ent(),ref.file(),"(",ref.line(),",",ref.column(),")")
  print ("\n",end="")
```

```
List of Functions with Parameters
---------------------------------
```

```
import understand

def sortKeyFunc(ent):
  return str.lower(ent.longname())

db = understand.open("test.udb")

ents = db.ents("function,method,procedure")
for func in sorted(ents,key = sortKeyFunc):
  print (func.longname()," (",sep="",end="")
  first = True
  for param in func.ents("Define","Parameter"):
    if not first:
      print (", ",end="")
    print (param.type(),param,end="")
    first = False
  print (")")
```

```
List of Functions with Associated Comments
------------------------------------------
```

```
import understand

db = understand.open("test.udb")

for func in db.ents("function ~unresolved ~unknown"):
  comments = func.comments("after")
  if comments:
    print (func.longname(),":\n  ",comments,"\n",sep="")
```

```
List of Ada Packages
--------------------
```

```
import understand

db = understand.open("test.udb")
```

```
print ("Standard Packages:")
for package in db.ents("Package"):
  if package.library() == "Standard":
    print ("  ",package.longname())

print ("\nUser Packages:")
for package in db.ents("Package"):
  if package.library() != "Standard":
    print("  ",package.longname())
```

```
All Project Metrics
-------------------

import understand

db = understand.open("test.udb")

metrics = db.metric(db.metrics())
for k,v in sorted(metrics.items()):
  print (k,"=",v)
```

```
Cyclomatic Complexity of Functions
----------------------------------

import understand

db = understand.open("test.udb")

for func in db.ents("function,method,procedure"):
  metric = func.metric(("Cyclomatic",))
  if metric["Cyclomatic"] is not None:
    print (func," = ",metric["Cyclomatic"],sep="")
```

```
"Called By" Graphs of Functions
-------------------------------

import understand

db = understand.open("test.udb")

for func in db.ents("function,method,procedure"):
  file = "callby_" + func.name() + ".png"
  print (func.longname(),"->",file)
  func.draw("Called By",file)
```

```
Info Browser View of Functions
------------------------------

import understand

db = understand.open("test.udb")

for func in db.ents("function,method,procedure"):
  for line in func.ib():
    print(line,end="")
```

```
Lexical Stream
--------------

import understand
```

```
db = understand.open("test.udb")

file = db.lookup("test.cpp")[0]
for lexeme in file.lexer():
  print (lexeme.text(),end="")
  if lexeme.ent():
    print ("@",end="")
```

Classes

builtins.Exception(builtins.BaseException)

understand.UnderstandError

builtins.object

understand.Arch
understand.Atn
understand.CFGraph
understand.CFNode
understand.Check
understand.Db
understand.Ent
understand.Kind
understand.Lexeme
understand.Lexer
understand.LexerIter
understand.Metric
understand.Option
understand.Ref
understand.Violation

class **Arch**(builtins.object)

```
This class represents an understand Architecture. Available methods are:

understand.Arch.children
understand.Arch.contains(entity [,recursive])
understand.Arch.depends(recursive=true,group=false)
understand.Arch.dependsby(recursive=true,group=false)
understand.Arch.draw(graph,filename [,options [,variant]])
understand.Arch.ents([recursive])
understand.Arch.longname()
understand.Arch.name()
understand.Arch.parent()  understand.Arch.__repr__()  --longname
understand.Arch.__str__()  --name
```

Methods defined here:

__repr__(self, /)

```
Return repr(self).
```

__str__(self, /)

```
Return str(self).
```

children(...)

```
arch.children() -> list of understand.Arch

Return the children of the architecture.
```

contains(...)

```
arch.contains(entity [,recursive]) -> bool

Return true if the entity is contained in the architecture

The parameter entity should be an instance of understand.Ent.

The optional parameter recursive specifies if the search is recursive.
If true, all nested architectures will be considered as well. It is
false by default.
```

depends(...)

```
arch.depends(recursive,group) ->
  dict key=understand.Arch value=list of understand.Ref

Return the dependencies of the architecture.

The optional parameter recursive is true by default. When false, child
architecture dependencies are not included.

The optional parameter group is false by default. When true, the keys
in the dictionary will be grouped into as few keys as possible.

For example, given the architecture structure:
  All
    Bob
      Lots of entities
    Sue
      Current
        Lots of entities
      Old
        Lots of entities

calling sue.depends(recursive=false) would return an empty dictionary
since sue's children (current and old) are not considered. Calling
bob.depends(group=true) would result in a single key in the
dictionary (Sue), as opposed to two keys (Sue/Current and Sue/Old)
since all the entities were grouped together.
```

dependsby(...)

```
arch.dependsby(recursive,group) ->
  dict key=understand.Arch value=list of understand.Ref

Return the architectures depended on by the architecture.

The optional parameter recursive is true by default. When false, child
architecture dependencies are not included.

The optional parameter group is false by default. When true, the keys
in the dictionary will be grouped into as few keys as possible.

For more information, see the help for understand.Arch.depends()
```

draw(...)

```
arch.draw(graph, filename [,options [,variant]]) -> None

Generate a graphics file for the architecture.
This command is not supported when running scripts
through the command line tool und
```

```
The parameter graph(string) should be the name of the graph to
generate. Available graphs vary by language and architecture, but the name
will be the same as the name in the Understand GUI. Some examples are:
  "Cluster Call"
  "Graph Architecture"
  "Internal Dependencies"
The parameter filename(string) should be the name of the file.
The filename must end with a supported extension (jpg, png, svg, vdx).
Some graphs support dot file format. Exporting a .dot file will also
export a svg image file.

The parameter options (string) is used to specify paramters used to
generate the graphics. The format of the options string is
"name=value". Multiple options are seperated with a semicolon.
spaces are allowed and are significant between mutli-word field names,
whereas, case is not significant. The valid names and values are the
same as appear in that graphs right click menu and vary by view. They
may be abbreviated to any unique prefix of their full names.

The parameter variant(string) is used to select from the available
variants for the graph. If a variant is not specified, then a default
variant will be used.

If an error occurs, and UnderstandError will be thrown.
Some possible errors
are:
  NoFont               - no suitable font can be found
  NoImage              - no image is defined or is empty
  NoVisioSupport       - no Visio .vsd files can be generated on
                         non-windows
  TooBig               - jpg does not support a dimension greater
                         than 64k
  UnableCreateFile     - file cannot be opened/created
  UnsupportedFile      - only .jpg, .png, or .svg files are supported
Additional error messages are also possible when generating a Visio
file.
```

ents(...)

```
arch.ents([recursive]) -> list of understand.Ent

Return the entities within the architecture.

The optional parameter recursive determines if nested architectures
are considered. It is false by default.
```

longname(...)

```
arch.longname() -> string

Return the long name of the architecture.
```

name(...)

```
arch.name() -> string

Return the short name of the architecture.
```

parent(...)

```
arch.parent() -> understand.Arch

Return the parent of the Arch or None if it is a root.
```

class **Atn**(builtins.object)

This class represents an understand annotation. Annotations can be obtained
for a database (db.annotations()) or for an entity (ent.annotations())
Available Methods are:
 understand.Atn.author()
 understand.Atn.ent()
 understand.Atn.date()
 understand.Atn.text()

Methods defined here:

author(...)
atn.author() -> string

Return the author who created the annotation.

date(...)
atn.date() -> string

Return the date the annotation was last modified as a string of the form
YYYY-MM-DDTHH:MM:SS such as 2000-01-01T19:20:30.

ent(...)
atn.ent() -> understand.Ent

Return the entity this annotation belongs to. This may be None if the
annotation is an orphan.

line(...)
atn.line() -> int

Return the line number (in the file) of the annotation or -1 if the
annotation is not a line annotation.

text(...)
atn.text() -> string

Return the text of the annotation.

class **CFGraph**(builtins.object)

A CFGraph is a control flow graph representation containing
CFNodes (control flow nodes). The start node and the list of all nodes
can be retrieved. The edges out of each node are given with the children
function. Some nodes may be unreachable from the start node. Available
methods are:

 understand.CFGraph.nodes()
 understand.CFGraph.start()

Methods defined here:

nodes(...)
CFGraph.nodes() -> list of understand.CFNode

Return a list of all nodes in the graph

start(...)
CFGraph.start() -> understand.CFNode

Return the start node of the graph.

class **CFNode**(builtins.object)

```
A node in a control flow graph. Available methods are:

understand.CFNode.child_label()
understand.CFNode.children()
understand.CFNode.column_begin()
understand.CFNode.column_end()
understand.CFNode.end_node()
understand.CFNode.kind()
understand.CFNode.line_begin()
understand.CFNode.line_end()
```

Methods defined here:

__eq__(self, value, /)

```
Return self==value.
```

__ge__(self, value, /)

```
Return self>=value.
```

__gt__(self, value, /)

```
Return self>value.
```

__le__(self, value, /)

```
Return self<=value.
```

__lt__(self, value, /)

```
Return self<value.
```

__ne__(self, value, /)

```
Return self!=value.
```

child_label(...)

```
CFNode.child_label(child) -> string

Return the label from this node to the given child CFNode or
None if none. Most nodes do not have labelled children. Possible
labels are:

  "yes": conditionals/loops
  "no": conditionals/loops
  "<0": fortran-arith-if
  "=0": fortran-arith-if
  ">0": fotran-arith-if
  "default": fotran-computed-goto, fotran-io-control
  "end" : fotran-io-control
  "eor" : fotran-io-control
  "err" : fotran-io-control
  "0","1",etc: fotran-computed-goto
```

children(...)

```
CFNode.children([flags]) -> list of understand.CFNode

Return the children of this node. If the optional parameter
flags is given, it should be a combination of the following
control flow node flags

  understand.CFNode_Normal
  understand.CFNode_Deferred
```

If flags are not provided, normal children are returned.

column_begin(...)

CFNode.column_begin() -> int

Return the beginning column number of the node.
Nodes without location information will return None

column_end(...)

CFNode.column_begin() -> int

Return the ending column number of the node.
Nodes without location information will return None.
The end column is inclusive

end_node(...)

CFNode.end_node() -> CFNode

Return the end node for structures such as loops or
conditionals that have an end, or None if there is
no associated end node.

kind(...)

cfnode.kind() -> string

Return the kind of the node. Possible kinds are:

ada-block-begin	end-repeat-until	fortran-select-case
basic-continue-do	end-routine	fortran-stop
basic-continue-for	end-select	fortran-until
basic-continue-while	end-switch	fortran-where
basic-do	end-try	goto
basic-do-until	end-with-do	if
basic-do-while	exception-when	java-block-begin
basic-end-do-loop	exit	jovial3-goto
basic-end-do-loop-until	exit-when	loop
basic-end-do-loop-while	for	next
basic-exit-do	fortran-arith-if	next-when
basic-exit-for	fortran-assigned-goto	passive
basic-exit-select	fortran-call-alt-return	passive-implicit
basic-exit-try	fortran-case	python-try-else
basic-exit-while	fortran-case-default	python-while-for-else
break	fortran-computed-goto	question-begin
case	fortran-cycle	question-colon
case-fallthru	fortran-do	raise
case-when	fortran-do-infinite	repeat-until
conditional-goto	fortran-do-while	return
continue	fortran-else-where	select
deferred-break	fortran-else-where-cond	select-else
deferred-continue	fortran-end-do	select-or
deferred-goto	fortran-end-select	select-then-abort
deferred-return	fortran-end-where	start
deferred-throw	fortran-exit-do	switch
do-while	fortran-exit-do-if	switch-case
else	fortran-exit-for	switch-default
elsif	fortran-exit-for-if	terminate
end	fortran-exit-if	throw
end-ada-block	fortran-exit-if-if	try
end-case	fortran-exit-loop	try-catch
end-do-while	fortran-exit-loop-if	try-finally
end-if	fortran-exit-while	while
end-java-block	fortran-exit-while-if	while-for
end-loop	fortran-io-control	with-do

line_begin(...)
```
CFNode.line_begin() -> int

Return the beginning line number of the node or
None if there is no associated text range.
```

line_end(...)
```
CFNode.line_end() -> int

Return the ending line number of the node or
None if there is no associated text range.
```

Data and other attributes defined here:

__hash__ = None

class **Check**(builtins.object)
```
Available Methods are:
understand.Check.db()
understand.Check.files()
understand.Check.is_aborted()
understand.Check.violation(entity,file,line,column,text)
understand.Check.option()
```

Methods defined here:

db(...)
```
check.db() -> understand.Db

Return the database associated with this check.
```

files(...)
```
check.files() -> list of understand.Ent

Return the list of files associated with this check.
```

id(...)
```
check.id() -> string

Return the id of this check.
```

is_aborted(...)
```
check.is_aborted() -> bool

Return True if the check has been aborted by the user.
```

option(...)
```
check.option() -> understand.Option

Return the Option object associated with this check.
```

violation(...)
```
check.violation(entity,file,line,column,text) -> understand.Violation

Emit a violation of this check.
```

class **Db**(builtins.object)

```
This class represents an understand database. With the exception of
Db.close(), all methods require an open database. A database is
opened through the module function understand.open(dbname). Available
methods are:

 understand.Db.add_annotation_file(path)
 understand.Db.annotations()
 understand.Db.archs(ent)
 understand.Db.close()
 understand.Db.comparison_db()
 understand.Db.ent_from_id(id)
 understand.Db.ents([kindstring])
 understand.Db.language()
 understand.Db.lookup(name [,kindstring])
 understand.Db.lookup_arch(longname)
 understand.Db.lookup_uniquename(uniquename)
 understand.Db.metric(metriclist)
 understand.Db.metrics()
 understand.Db.metrics_treemap(file, sizemetric, colormetric [enttype [,arch]])
 understand.Db.name()
 understand.Db.relative_file_name()
 understand.Db.root_archs()  understand.Db.__str__() --name
```

Methods defined here:

__str__(self, /)

```
Return str(self).
```

add_annotation_file(...)

```
db.add_annotation_file(path [,foreground [,background]]) -> None

Add a new or existing annotation database file to this database.
The added file is set as the currently selected annotation database.
The foreground and background arguments should take the form #RRGGBB.
```

annotations(...)

```
db.annotations() -> list of understand.Atn

Return a list of annotations for the database.
```

archs(...)

```
db.archs(ent) -> list of understand.Arch

Return a list of architectures that contain ent (understand.Ent)
```

close(...)

```
db.close() -> None

Close the database.

This allows a new database to be opened. It will never throw an
error and is safe to call even if the database is already closed.
After the database is closed, accessing objects associated with
the database (ents, refs, ...) can cause Python to crash.
```

comparison_db(...)

```
db.comparison_db() -> understand.Db

Return the comparison database associated with this database.
```

ent_from_id(...)

db.ent_from_id(id) -> understand.Ent

Return the ent associated with the id.

The id is obtained using ent.id. This should only be called for identifiers that have been obtained while the database has remained open. When a database is reopened, the identifier is not guaranteed to remain consistent and refer to the same entity.

ents(...)

db.ents([kindstring]) -> list of understand.Ent

Return a list entities in the database.

If the optional parameter kindstring(string) is not passed, then all the entities in the database are returned. Otherwise, kindstring should be a language-specific entity filter string. The database must be open or a UnderstandError will be thrown.

language(...)

db.language() -> tuple of strings

Return a tuple with project languages

This method returns a tuple containing all the language names enabled in the project. Possible language names are: "Ada", "C++", "C#", "Fortran", "Java", "Jovial", "Pascal", "Plm", "Python", "VHDL", or "Web". C is included with "C++"
This will throw a UnderstandError if the database has been closed.

lookup(...)

db.lookup(name [,kindstring]) -> list of understand.Ent

Return a list of entities that match the specified name.

The parameter name should be a regular expression, either compiled or as a string. By default, regular expressions are case sensitive. For case insensitive search, compile the regular expression like this:

```
import re
db.lookup(re.compile("searchstring",re.I))
```

The re.I flag is for case insensitivity. Otherwise, the lookup command can be run simply

```
db.lookup("searchstring")
```

The optional paramter kindstring is a language-specific entity filter string. So, for example,

```
db.lookup(".Test.","File")
```

would return a list of file entities containing "Test" (case sensitive) in their names.

lookup_arch(...)

db.lookup_arch(longname) -> understand.Arch

Return the architecture with the given longname, or None if not found.

lookup_uniquename(...)

db.lookup_uniquename(uniquename) -> ent

Return the entity identified by uniquename.

Uniquename is the name returned by ent.uniquename and repr(ent). This will return None if no entity is found.

metric(...)

db.metric(metriclist) -> dict key=string value=metricvalue

Return the metric value for each item in metriclist

Metric list must be a tuple or list containing the names of metrics
as strings. If the metric is not available, it's value will be None.

metrics(...)

db.metrics() -> list of strings

Return a list of project metric names.

metrics_treemap(...)

db.metrics_treemap(file, sizemetric, colormetric [,enttype [,arch]]) -
> None

Export a metrics treemap to the given file (must be jpg
or png). The parameters
sizemetric and colormetric should be the API names of
the metrics. The optional
parameter arch is the group-by arch. If none is given,
the graph will be flat.
The optional parameter enttype is the type of entities to use in the
treemap. It
must be a string either "file" "class" or "function". If none is given,
file is assumed.

name(...)

db.name() -> string

Return the filename of the database.

This will throw a UnderstandError if the database has been closed.

relative_file_name(...)

db.relative_file_name(absolute_path) -> string

Return the relative file name like ent.relname() but for an
arbitrary path.

root_archs(...)

db.root_archs() -> list of understand.Arch

Return the root architectures for the database.

class **Ent**(builtins.object)

This class represents an understand entity(files, functions,
variables, etc). Available methods are:

understand.Ent.annotate(text [,offset])
understand.Ent.annotations()
understand.Ent.comments([style [,raw [,refkindstring]]])
understand.Ent.contents()
understand.Ent.control_flow_graph()
understand.Ent.depends()
understand.Ent.dependsby()
understand.Ent.draw(graph,filename [,options [,variant]])
understand.Ent.ents(refkindstring [,entkindstring])
understand.Ent.__eq__() --by id
understand.Ent.filerefs([refkindstring [,entkindstring [,unique]]])
understand.Ent.__ge__() --by id

```
  understand.Ent.__gt__() --by id
  understand.Ent.__hash__() --id
  understand.Ent.ib([options])
  understand.Ent.id()
  understand.Ent.kind()
  understand.Ent.kindname()
  understand.Ent.language()
  understand.Ent.__le__() --by id
  understand.Ent.lexer([lookup_ents [,tabstop [,show_
inactive [,expand_macros]]]])
  understand.Ent.library()
  understand.Ent.longname()
  understand.Ent.__lt__() --by id
  understand.Ent.metric(metriclist)
  understand.Ent.metrics()
  understand.Ent.name()
  understand.Ent.__ne__() --by id
  understand.Ent.parameters(shownames = True)
  understand.Ent.parent()
  understand.Ent.parsetime()
  understand.Ent.ref([refkindstring [,entkindstring]])
  understand.Ent.refs([refkindstring [,entkindstring [,unique]]])
  understand.Ent.relname()
  understand.Ent.__repr__() --uniquename
  understand.Ent.simplename()
  understand.Ent.__str__() --name
  understand.Ent.type()
  understand.Ent.uniquename()
  understand.Ent.value()
```

Methods defined here:

__eq__(self, value, /)
    Return self==value.

__ge__(self, value, /)
    Return self>=value.

__gt__(self, value, /)
    Return self>value.

__hash__(self, /)
    Return hash(self).

__le__(self, value, /)
    Return self<=value.

__lt__(self, value, /)
    Return self<value.

__ne__(self, value, /)
    Return self!=value.

__repr__(self, /)
    Return repr(self).

__str__(self, /)
    Return str(self).

annotate(...)
    ent.annotate(text [,author [,offset]]) -> None

Add text as a new annotation associated with this entity.
The annotation is added to the current annotation database
with the currently set user name.

annotations(...)

ent.annotations() -> list of understand.Atn

Return the annotations associated with the entity, or empty list if
there are none.

comments(...)

ent.comments([style [,raw [,refkindstring]]]) -> string

Return the comments associated with the entity.

The optional paramter style (string) is used to specify which comments
are to be used. By default, comments that come after the entity
declaration are processed. Possible values are:

default	- same as after
after	- process comments after the entity declaration
before	- process comments before the entity declaration

If a different value is passed in, it will be silently ignored.

The optional paramater raw (true/false) is used to specify what kind of
formatting, if any, is applied to the comment text. If raw is false,
function will remove comment characters and certain repeating
characters, while retaining the original newlines. If raw is true, the
function will return a list of comment strings in original format,
including comment characters.

The optional parameter refkindstring should be a language specific
reference filter string. For C++, the default is "definein",
which is almost always correct. However, to see comments associated
member declarations, "declarein" should be used. For Ada, there
are many declaration kinds that may be used, including "declarein
body", "declarein spec" and "declarein instance". A bad
refkindstring may result in an UnderstandError.

contents(...)

ent.contents() -> string

Return the contents of the entity.

Only certain entities are supported, such as files and defined
functions. Entities with no contents will return empty string.

control_flow_graph(...)

ent.control_flow_graph() -> understand.CFGraph

Return a control flow graph for the entity.

Only certain entities are supported. If no control information is
known, then None will be returned.

depends(...)

ent.depends() -> dict key=understand.Ent value=list of understand.Ref

Return the dependencies of the class or file

This function returns all the dependencies as a dictionary between an
ent and the references occurring in the ent. An empty dictionary will
be returned if there are no dependencies for the ent. The ent should be
a class or file.

dependsby(...)

```
ent.dependsby() -> dict key=understand.Ent value=list of understand.Ref

Return the ents depended on by the class or file

This function returns all the dependencies as a dictionary between an
ent and the references occurring in the ent. An empty dictionary will
be returned if there are no dependencies on the ent. The ent should be
a class or file.
```

draw(...)

```
ent.draw(graph, filename [,options [,variant]]) -> None

Generate a graphics file for the entity

The parameter graph(string) should be the name of the graph to
generate. Available graphs vary by language and entity, but the name
will be the same as the name in the Understand GUI. Some examples are:
  "Base Classes"
  "Butterfly"
  "Called By"
  "Control Flow"
  "Calls"
  "Declaration"
  "Depends On"

The parameter filename(string) should be the name of the file.
The filename must end with a supported extension (jpg, png, svg, vdx).
Some graphs support dot file format. Exporting a .dot file will also
export a svg image file.

The parameter options (string) is used to specify paramters used to
generate the graphics. The format of the options string is
"name=value". Multiple options are seperated with a semicolon.
spaces are allowed and are significant between mutli-word field names,
whereas, case is not significant. The valid names and values are the
same as appear in that graphs right click menu and vary by view. They
may be abbreviated to any unique prefix of their full names. Some
examples are:
  "Layout=Crossing; name=Fullname;Level=AllLevels"
  "Display Preceding Comments=On;Display Entity Name=On"
For Relationship graphs use  secondent=EntityUniqueName to
indicate the second entity

The parameter variant(string) is used to select from the available
variants for the graph. If a variant is not specified, then a default
variant will be used.

If an error occurs, and UnderstandError will be thrown.
Some possible errors
are:
  NoFont               - no suitable font can be found
  NoImage              - no image is defined or is empty
  NoVisioSupport       - no Visio .vsd files can be generated on
                         non-windows
  TooBig               - jpg does not support a dimension greater
                         than 64k
  UnableCreateFile     - file cannot be opened/created
  UnsupportedFile      - only .jpg, .png, or .svg files are supported
Additional error messages are also possible when generating a Visio
file.
```

ents(...)

```
ent.ents(refkindstring [,entkindstring]) -> list of understand.Ent

Return a list of entities that reference, or are
referenced by, the entity.

The parameter refkindstring (string) should be a language-specific
reference filter string.

The optional paramater entkindstring (string) should be a language-
specific entity filter string that specifies what kind of referenced
entities are to be returned. If it is not included, all referenced
entities are returned.
```

filerefs(...)

```
ent.filerefs([refkindstring [,entkindstring [,unique]]]) ->
list of understand.Ref

Return a list of all references that occur in a file entity.

If this is called on a non-file entity, it will return an empty list.
The references returned will not necessarily have the file entity for
their .scope value.

The optional paramter refkindstring (string) should be a language-
specific reference filter string. If it is not given, all references
 are returned.

The optional paramter entkindstring (string) should be a language-
specific entity filter string that specifies what kind of referenced
entities should be returned. If it is not given, all references to
any kind of entity are returned.

The optional parameter unique (bool) is false by default. If it is
true, only the first matching reference to each unique entity is
returned
```

freetext(...)

```
ent.freetext(option) -> string
```

ib(...)

```
ent.ib([options]) -> list of strings

Return the Info Browser information for an entity.

The optional parameter options (string) may be used to specify some
parameters used to create the text. The format of the options string
is "name=value" or "{field-name}name=value". Multiple options are
separated with a semicolon. Spaces are allowed and are significant
between multi-word field names, whereas, case is not significant. An
option that specifies a field name is specific to that named field of
the Info Browser. The available field names are exactly as they appear
in the Info Browser. When a field is nested within another field, the
correct name is the two names combined. For example, in C++, the field
Macros within the field Local would be specified as "Local Macros".

A field and its subfields may be disabled by specifying levels=0, or
by specifying the field off, without specifying any option. For example,
either of the will disable and hide the Metrics field:
  {Metrics}levels=0;
  {Metrics}=off;
The following option is currently available only without a field name.
  Indent    - this specifies the number of indent spaces to output for
              each level of a line of text. The default is 2.
```

```
Other options are the same as are displayed when right-clicking on the
field name in the Understand tool. No defaults are given for these
options, as the defaults are specific for each language and each field
name
An example of a properly formatted option string would be:
  "{Metrics}=off;{calls}levels=-1;{callbys}levels=-1;{
references}sort=name"

The Architectures field is not generated by this command and can be
generated separately using db.archs(ent)
```

id(...)

```
ent.id() -> int

Return a unique numeric identifier for the entity.

The identifier is not guaranteed to remain constant after the
database has been updated. An id can be converted back into an
understand.Ent with db.ent_from_id(id). The id is used for
comparisons and the hash function.
```

kind(...)

```
ent.kind() -> understand.Kind

Return the kind object for the entity.
```

kindname(...)

```
ent.kindname() -> string

Return the simple name for the kind of the entity.

This is similar to ent.kind().name(), but does not create a Kind
object.
```

language(...)

```
ent.language() -> string

Return the language of the entity

Possible values include "Ada", "C++","C#", "Fortran",
"Java", "Jovial", "Pascal", "Plm", "Python",
"VHDL" or "Web". C is included with "C++".
```

lexer(...)

```
ent.lexer([lookup_ents [,tabstop [,show_inactive [,expand_macros]]]])
  -> understand.Lexer

Return a lexer object for the specified file entity. The original
source file must be readable and unchanged since the last database
parse. If an error occurs, an UnderstandError will be thrown. Possible
errors are:
  FileModified         - the file must not be modified since the last
                         parse
  FileUnreadable       - the file must be readable from the original
                         location
  UnsupportedLanguage  - the file language is not supported

The optional paramter lookup_ents is true by default. If it is
specified false, the lexemes for the constructed lexer will not
have entity or reference information, but the lexer construction will
be much faster.

The optional paramter tabstop is 8 by default. If it is specified it
must be greater than 0, and is the value to use for tab stops
```

The optional parameter show_inactive is true by default. If false,
inactive lexemes will not be returned.

The optional parameter expand_macros is false by default. If true,
and if macro expansion text is stored, lexemes that are macros will
be replaced with the lexeme stream of the expansion text.

library(...)

ent.library() -> string

Return the library the entity belongs to.

This will return "" if the entity does not belong to a
library. Predefined Ada entities such as text_io will bin the
'Standard' library. Predefined VHDL entities will be in either the
'std' or 'ieee' libraries.

longname(...)

ent.longname() -> string

Return the long name of the entity.

If there is no long name defined, the regular name (ent.name()) is
returned. Examples of entities with long names include files, c++
members, and most ada entities.

metric(...)

ent.metric(metriclist) -> dict key=string value=metricvalue

Return the metric value for each item in metriclist

Metric list must be a tuple or list containing the names of metrics
as strings. If the metric is not available, it's value will be None.

metrics(...)

ent.metrics() -> list of strings

Return a list of metric names defined for the entity.

name(...)

ent.name() -> string

Return the shortname for an entity.

For Java, this may return a name with a single dot in it. Use
ent.simplename() to obtain the simplest, shortest name possible. This
is what str() shows.

parameters(...)

ent.parameters(shownames=True) -> string

Return a string containing the parameters for the entity.

The optional parameter shownames should be True or False. If it is
False only the types, not the names, of the parameters are returned.
There are some language-specific cases where there are no entities in
the database for certain kinds of parameters. For example, in c++,
there are no database entities for parameters for functions that are
only declared, not defined, and there are no database entities for
parameters for functional macro definitions. This method can be used
to get some information about these cases. If no parameters are
available, None is returned.

parent(...)

```
ent.parent() -> understand.Ent

Return the parent of the entity or None if none
```

parsetime(...)

```
ent.parsetime() -> int

Return the last time the file entity was parse in the database.

If the entity is not a parse file, it will be 0. The time is in
Unix/Postix Time
```

ref(...)

```
ent.ref([refkindstring [,entkindstring]) -> understand.Ref

This is the same as ent.refs()[:1]
```

refs(...)

```
ent.refs([refkindstring [,entkindstring [,unique]]]) ->
list of understand.Ref

Return a list of references.

The optional paramter refkindstring (string) should be a language-
specific reference filter string. If it is not given, all references
 are returned.

The optional paramter entkindstring (string) should be a language-
specific entity filter string that specifies what kind of referenced
entities should be returned. If it is not given, all references to
any kind of entity are returned.

The optional parameter unique (bool) is false by default. If it is
true, only the first matching reference to each unique entity is
returned
```

relname(...)

```
ent.relname() -> string

Return the relative name of the file entity.

This is the fullname for the file, minus any root directories that
are common for all project files. Return None for non-file entities.
```

simplename(...)

```
ent.simplename() -> string

Return the simplename for the entity.

This is the simplest, shortest name possible for the entity. It is
generally the same as ent.name() except for languages like Java, for
which this will not return a name with any dots in it.
```

type(...)

```
ent.type() -> string

Return the type string of the entity.

This is defined for entity kinds like variables and types, as well as
entity kinds that have a return type like functions.
```

uniquename(...)

ent.uniquename() -> string

Return the unique name of the entity.

This name is not suitable for use by an end user. Rather, it is a means of identifying an entity uniquely in multiple databases, perhaps as the source code changes slightly over time. The unique name is composed of things like parameters and parent names. So, the some code changes will in new uniquenames for the same intrinsic entity. Use db.lookup_uniquename() to convert a unqiuename back to an object of understand.Ent. This is what repr() shows.

value(...)

ent.value() -> string

Return the value associated with the entity.

This is for enumerators, initialized variables, and macros. Not all languages are supported.

class **Kind**(builtins.object)

This class represents a kind of an entity or reference.
For example,an entity kind might be a "C Header File" and a reference kind kind could be "Call." Kindstrings and refkindstrings filters are built from these. A filter string may use the tilde "~" to indicate the absence of a token, and comma "," to "or" filters together. Otherwise, filters are constructed with an "and" relationship. For more information on filter strings or a full list of available kinds and reference kinds see the Understand Perl API documentation.

Available methods are:

```
  understand.Kind.check(kindstring)
  understand.Kind.inv()
  understand.Kind.longname()
  understand.Kind.name()  understand.Kind.__repr__() --longname
  understand.Kind.__str__() --name
Static Methods:
  understand.Kind.list_entity([entkind])
  understand.Kind.list_reference([refkind])
```

Methods defined here:

__eq__(self, value, /)
Return self==value.

__ge__(self, value, /)
Return self>=value.

__gt__(self, value, /)
Return self>value.

__hash__(self, /)
Return hash(self).

__le__(self, value, /)
Return self<=value.

__lt__(self, value, /)
Return self<value.

__ne__(self, value, /)
Return self!=value.

__repr__(self, /)
Return repr(self).

__str__(self, /)
Return str(self).

check(...)
kind.check(kindstring) -> bool

Return true if the kind matches the filter string kindstring.

inv(...)
kind.inv() -> understand.Kind

The logical inverse of a reference kind. This will throw an
UnderstandError if called with an entity kind.

longname(...)
kind.longname() -> string

Return the long form of the kind name.

This is usually more detailed than desired for human reading. It is
the same as repr(kind)

name(...)
kind.name() -> string

Return the name of the kind.

This is the same as str(kind).

Static methods defined here:

list_entity(...)
Kind.list_entity([entkind]) (static method)-> list of understand.Kind

Return the list of entity kinds that match the filter entkind.

If no entkind is given, all entity kinds are returned. For example,
to get the list of all c function entity kinds:
  kinds = understand.Kind.list_entity("c function")

list_reference(...)
Kind.list_reference([refkind]) (static method)->list of understand.Kind

Return the list of reference kinds that match the filter refkind.

If no refkind is given, all reference kinds are returned. For example,
to get the list of all ada declare reference kinds:
  kinds = understand.Kind.list_entity("ada declare")

class **Lexeme**(builtins.object)

A lexeme is basically a token recieved from an Lexer. Available
methods are:

```
understand.Lexeme.column_begin()
understand.Lexeme.column_end()
understand.Lexeme.ent()
understand.Lexeme.inactive()
understand.Lexeme.line_begin()
understand.Lexeme.line_end()
understand.Lexeme.next()
understand.Lexeme.previous()
understand.Lexeme.ref()
understand.Lexeme.text()
understand.Lexeme.token()
```

Methods defined here:

column_begin(...)

```
lexeme.column_begin() -> int

Return the beginning column number of the lexeme.
```

column_end(...)

```
lexeme.column_end() -> int

Return the ending column number of the lexeme.
```

ent(...)

```
lexeme.ent() -> Understand.Ent

Return the entity associated with the lexeme or None if none.
```

inactive(...)

```
lexeme.inactive() -> bool

Return True if the lexeme is part of inactive code.
```

line_begin(...)

```
lexeme.line_begin() -> int

Return the beginning line number of the lexeme.
```

line_end(...)

```
lexeme.line_end() -> int

Return the ending line number of the lexeme.
```

next(...)

```
lexeme.next([ignore_whitespace [,ignore_comments]]) -> understand.Lexeme

Return the next lexeme, or None if no lexemes remain.
```

previous(...)

```
lexeme.previous([ignore_whitespace [,ignore_comments]]) -
> understand.Lexeme

Return the previous lexeme, or None if beginning of file.
```

ref(...)

```
lexeme.ref() -> understand.Ref

Return the reference associated with the lexeme, or None if none.
```

text(...)

```
lexeme.text() -> string

Return the text for the lexeme, which may be empty ("").
```

token(...)

```
lexeme.token() -> string

Return the token kind of the lexeme.

Values include:
  Comment
  Continuation
  EndOfStatemen
  Identifier
  Keyword
  Label
  Literal
  Newline
  Operator
  Preprocessor
  Punctuation
  String
  Whitespace
  Indent
  Dedent
```

class **Lexer**(builtins.object)

```
A lexer is a lexical stream generated for a file entity, if the
original file exists and is unchanged from the last database reparse.
The first lexeme (token) in the stream can be accessed using
Lexer.first. A lexer can be iterated over, where each item returned is
a lexeme. Available methods are:

  understand.Lexer.first()
  understand.Lexer.__iter__()
  undersatnd.Lexer.lexeme(line,column)
  understand.Lexer.lexemes([start_line [,end_line]])
  understand.Lexer.lines()
```

Methods defined here:

__iter__(self, /)

```
Implement iter(self).
```

first(...)

```
lexer.first() -> lexeme

Return the first lexeme for the lexer.
```

lexeme(...)

```
lexer.lexeme(line,column) -> understand.Lexeme

Return the lexeme at the specified line and column.
```

lexemes(...)

```
lexer.lexemes([start_line [,end_line]]) -> list of understand.Lexeme

Return all lexemes. If the optional parameters start_line or end_line
are specified, only the lexemes within these lines are returned.
```

lines(...)
```
lexer.lines() -> int

Return the number of lines in the lexer.
```

class **LexerIter**(builtins.object)

Methods defined here:

__iter__(self, /)
```
Implement iter(self).
```

__next__(self, /)
```
Implement next(self).
```

class **Metric**(builtins.object)
```
This class is really just a shell for the two methods, description
and list. The description method will give a description for a metric
and list will list all the available metrics.
```

Static methods defined here:

__new__(*args, **kwargs) from builtins.type
```
Create and return a new object.  See help(type) for accurate signature.
```

description(...)
```
Metric.description(metricname) (static method) -> string

Return a description of the metric.

The parameter metricname is the string name of the metric. This will
return an empty string if there is no metric for metricname
```

list(...)
```
Metric.list([kindstring]) (static method) -> list of strings

Return a list of metric names.

The optional parameter kindstring should be a filter string. If given
only the names of metrics defined for entities that match are returned.
Otherwise, all possible metric names are returned.
```

class **Option**(builtins.object)
```
Available Methods are:
understand.Option.checkbox(name,text,default)
understand.Option.choice(name,choices,default)
understand.Option.integer(name,text,default)
understand.Option.text(name,text,default)
understand.Option.lookup(name)
```

Methods defined here:

checkbox(...)

option.checkbox(name,text,default) -> None

Create a checkbox option.

choice(...)
option.choice(name,text,choices,default) -> None

Create a choice option.

integer(...)
option.integer(name,text,default) -> None

Create an integer option.

lookup(...)
option.lookup(name) -> Any

Lookup an option value by name.

text(...)
option.text(name,text,default) -> None

Create a text option.

class **Ref**(builtins.object)

A reference object stores an reference between on entity an another.
Available methods are:

understand.Ref.column()
understand.Ref.ent()
understand.Ref.file()
undersatnd.Ref.kind()
understand.Ref.kindname()
understand.Ref.line()
understand.Ref.scope()
understand.Ref.__str__() --kindname ent file(line)

Methods defined here:

__repr__(self, /)
Return repr(self).

__str__(self, /)
Return str(self).

column(...)
ref.column() -> int

Return the column in source where the reference occurred.

ent(...)
ref.ent() -> understand.Ent

Return the entity being referenced.

file(...)
ref.file() -> understand.Ent

Return the file where the reference occurred.

isforward(...)
```
ref.isforward() -> bool

Return True if the reference is forward.
```

kind(...)
```
ref.kind() -> understand.Kind

Return the reference kind.
```

kindname(...)
```
ref.kindname() -> string

Return the short name of the reference kind

This is similar to ref.kind().name(), but does not create
anunderstand.Kind object.
```

line(...)
```
ref.line() -> int

Return the line in source where the reference occurred.
```

macroexpansion(...)
```
ref.macroexpansion() -> string

Return the macro expansion text for the refence file, line and column.
This function may return None if no text is available.
```

scope(...)
```
ref.scope() -> understand.Ent

Return the entity performing the reference.
```

class **UnderstandError**(builtins.Exception)

Method resolution order:
UnderstandError
builtins.Exception
builtins.BaseException
builtins.object

Data descriptors defined here:

__weakref__
```
list of weak references to the object (if defined)
```

Methods inherited from builtins.Exception:

__init__(self, /, *args, **kwargs)
```
Initialize self.  See help(type(self)) for accurate signature.
```

Static methods inherited from builtins.Exception:

__new__(*args, **kwargs) from builtins.type
```
Create and return a new object.  See help(type) for accurate signature.
```

Methods inherited from builtins.BaseException:

__delattr__(self, name, /)
Implement delattr(self, name).

__getattribute__(self, name, /)
Return getattr(self, name).

__reduce__(...)
Helper for pickle.

__repr__(self, /)
Return repr(self).

__setattr__(self, name, value, /)
Implement setattr(self, name, value).

__setstate__(...)

__str__(self, /)
Return str(self).

with_traceback(...)
Exception.with_traceback(tb) --
set self.__traceback__ to tb and return self.

Data descriptors inherited from builtins.BaseException:

__cause__
exception cause

__context__
exception context

__dict__

__suppress_context__

__traceback__

args

class **Violation**(builtins.object)

```
Available Methods are:
understand.Violation.add_fixit_hint(line,column,length[,text])
```

Methods defined here:

add_fixit_hint(...)
violation.add_fixit_hint(line,column,end_line,end_column[,text]) -> None

Add a fix-it hint associated with this violation.

The line, column, end_line, and end_column describe a range of text to be replaced in the file. The range can be empty to indicate pure insertion. The text is the replacement text. It can be empty for pure removal.

Functions

checksum(...)

```
understand.checksum(text [,len]) -> string

Return a checksum of the text

The optional parameter len specifies the length of the checksum,
which may be between 1 and 32 characters, with 32 being the default
```

license(...)

```
understand.license(name) -> None

Set a regcode string or a specific path to an understand license
```

open(...)

```
understand.open(dbname) -> understand.Db

Open a database from the passed in filename.

This returns a new understand.Db given the dbname (string). It
will throw an understand.UnderstandError if unsuccessful. Possible causes
for error are:
  DBAlreadyOpen        - only one database may be open at once
  DBCorrupt            - bad database file
  DBOldVersion         - database needs to be rebuilt
  DBUnknownVersion     - database needs to be rebuilt
  DBUnableOpen         - database is unreadable or does not exist
  NoApiLicense         - Understand license required
```

version(...)

```
understand.version() -> int

Return the current build number for this module
```

Data

CFNode_Deferred = 2
CFNode_Normal = 1
COMMENT = 'Comment'
CONTINUATION = 'Continuation'
DEDENT = 'Dedent'
ENDOFSTATEMENT = 'EndOfStatement'
EOF = 'EOF'
IDENTIFIER = 'Identifier'
IDSEQ = 'IdSeq'
INDENT = 'Indent'
KEYWORD = 'Keyword'
LABEL = 'Label'
LITERAL = 'Literal'
NEWLINE = 'Newline'
OPERATOR = 'Operator'
PREPROCESSOR = 'Preprocessor'
PUNCTUATION = 'Punctuation'

STRING = 'String'
WHITESPACE = 'Whitespace'