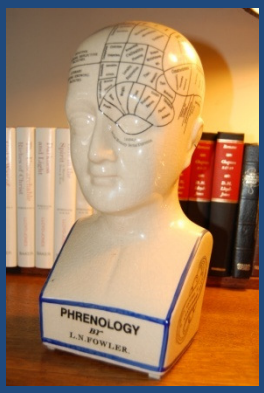

12/21/2009

Python ACT-R

User Guide version 1.0

Carleton University
Authored by: Kam Kwok

Chapter 1: Introduction

What we will discuss in this section:

- ✓ The purpose of the user guide.
- ✓ Prerequisites.
- ✓ Why Python ACT-R
- ✓ Getting started.

Introduction

The Purpose of This Guide

The aim of this guide is to provide a tutorial to readers who have no prior knowledge of Python ACT-R (pACT-R) and are interested in using Python ACT-R to build a cognitive model.

Prerequisites

The following prerequisites are provided for reference purposes. Readers who encounter difficulties in understanding this tutorial should consult these sites for help:

- basic Python syntax and programming knowledge (http://docs.python.org/tutorial/)
- ACT-R theory (http://actr.psy.cmu.edu/papers/403/IntegratedTheory.pdf)
- basic cognitive modeling principles (http://ccmsuite.ccmlab.ca/?q=node/2)

Useful Links

The followings are some resource sites that the reader should visit often. These sites contain most of the resources and references that you will need to start building your cognitive model.

Dr. Robert West's Python ACT-R course website:
http://sites.google.com/site/pythonactr/

All p-ACT-R codes are available on Dr. Robert West's Python ACT-R course site.

Dr. Terry Stewart's CCMLAB tutorial site:

http://ccmsuite.ccmlab.ca/

ACT-R main site at CMU:

http://act-r.psy.cmu.edu/

Why Python ACT-R

The primary motivations for re-implementing ACT-R core functions in Python are: 1. to verify that ACT-R theory is implementation independent and 2. to provide a more accessible programming interface for future ACT-R module development. The full rationale and results are discussed in Deconstructing ACT-R[1].

Getting Started

Install Python:

- To set up Python you need to download and install Python 2.6.2
- download the right version for your operating system and install: http://www.python.org/download/releases/2.6.2/

Install Python ACT-R (Install CCMSuite)

The CCMSuite is the programming library for Python ACT-R. To build a pACT-R model, you need to install the CCMSuite.

Follow the link below. Review the instructions and download the CCMSuite.zip file.

http://sites.google.com/site/pythonactr/set-up/ccmsuite-download/

[1] 2006, Stewart T., West R., Deconstructing ACT-R
http://sites.google.com/site/pythonactr/reference-material/deconstructing-act-r

Chapter 2: Background

What we will discuss in this section:

- ✓ What is ACT-R.
- ✓ ACT-R Core Components and pACT-R.
- ✓ Modeling in Python ACT-R.

What is ACT-R

The acronym ACT-R currently stands for the 'Adaptive Control of Thought –Rational', which is the evolutionary outcome from what was previously known as the 'Atomic Component of Thought' theory by John Anderson.

ACT-R theory is a cognitive architecture. A cognitive architecture is an integrated theory for cognition, which is supported by a computational framework. The computational framework provides a set of functional components that are functionally equivalent to the brain functions.

For more detailed information, please refer to the latest description of the ACT-R theory in the paper "An Integrated Theory of the Mind" (2004) which is available from the ACT-R web site at:
http://actr.psy.cmu.edu/papers/403/IntegratedTheory.pdf.

ACT-R Core Components

The core components of ACT-R theory consists of:

- The central executive function working with the goal buffer.
- Working memory buffers representing a mind state.
- Declarative memory module (semantic memory) –chunks.
- Production module - production rules[2].
- The sub-symbolic modules/parameters.

[2] Production rules are like "IF <conditions…>THEN <actions…>" rules.

The current Python ACT-R fully supports these core components described above. And it is the purpose of this user guide to show users how to build cognitive models from the core symbolic components.

Modeling in Python ACT-R

To build a pACT-R model is not the same as using Python to build an artificially intelligent program. To be an acceptable pACT-R model, only the core components identified above and which are provided by the framework to implement the cognitive tasks are to be used. The user's model must only use these buffers, declarative memory, and production rules for the agent as no other mechanisms will be permitted in your cognitive tasks unless they are grounded on a cognitive theory. i.e. no programming shortcuts should be used in a model unless it is a part of the theory that you are postulating.

Also, pACT-R is "white space sensitive" so be aware that the program recognizes each and every location as important data. So, your model MUST be sensitive to this requirement!

To build a basic cognitive model the user has to create:

- an agent which embodies the core components (mandatory)
- an environment module (necessary)
- a vision module (optional)
- a motor module (optional)

The advantage of using pACT-R is that it offers users accessible open components for users to build new modules in Python (for new theories), and users can seamlessly integrate the new modules with the core pACT-R.

Chapter 3: Modeling Elements

What we will discuss in this section:

- ✓ Models
- ✓ Agents.
- ✓ Environments.

Models

An ACT-R model is typically defined as one simulated cognitive agent. However, for users who are interested in building a multi-agents system, the implementation of a multi-agents system in Python ACT-R is a relatively straight forward programming exercise. And this should be noted as one of the advantages of using Python ACT-R.

An example of multi-agents system ("simple-agents") is available on Dr. Robert West's course website:

http://sites.google.com/site/pythonactr/perceptual-motor/simple-agents

Agents

The most central element of a user's model is the agent. A user model must have at least one agent. An agent is an actor or an active element for the theory[3]. The agent(s) embodies the specification of the cognitive task that the user is modeling and the agent is built by using the core components provided in ACT-R.

[3] A 'theory' is defined as an explanation or description of a cognitive task based upon the operations of the ACT-R core components.

Defining an Agent

If you want to create an agent for your model, the first step is to define its class. The following Example 1 defines a user agent class called MyAgent (a 'class' is a template for creating an object).

Lines 1 and 2 are import statements for including the 'ccm' module for your model. The 'ccm' module provides all the necessary Python ACT-R classes for modeling. Note that all comments are to be preceded by a '#' sign.

Line 3 declares a user defined agent class called MyAgent, which is inherited from the CCMSuite ACTR class. An agent is created with a production module by default; the production module is a part of ACTR class. The user can define production rules as a part of the agent without any additional setup.

Notice, Line 4 is indented! This means line 4 is a sub-level to the previous statement (here, it means it is a part of the class definition).

Line 4 creates the focus buffer for the agent; by convention, the 'focus buffer' is used as the (nomenclature change) 'goal buffer' in pACT-R. The buffer a created by the "Buffer( )" function.

Line 5 defines the 'initialization production rule'– 'def init()' is the first production rule that an agent will execute when the agent object is instantiated[4] (created). In this example, the action of the rule is to set focus buffer to 'sandwich bread' using the object's '.set' method. In general, a buffer can be set by its '.set' method (i.e. <buffer-name>.set (<'value'>)).

The main purpose of the focus buffer is to represent the model's central executive attention function and it is used to guide or direct the flow of the cognitive task. Typically, inside a production rule, the focus buffer is set to the value of the buffer condition for the next step in the cognitive task. As in line 6, the focus buffer is set to 'sandwich bread', the 'agent is directed to the next rule that has its focus buffer condition equals to 'sandwich bread'. In this example, it is the production rule defined in line 7, the bread_bottom rule. Lines 8 and 9 define the actions of the bread_bottom production rule. Line 8 prints the string "I have a piece of bread" to the console, and line 9 again sets the focus buffer to the condition of the next rule for the task.

[4] The "instantiation" of an agent object is being described in the 'Environment' section.

Example 1: Defining a Python ACT-R Agent (Code Fragment)

```
1.  import ccm                          # import ccm module library for Python ACT-R classes
2.  from ccm.lib.actr import *

3.  class MyAgent(ACTR):                # Defining  an act-r agent of type MyAgent.

4.      focus=Buffer()                  # Creating the goal buffer for the agent

5.      def init():                     # this rule fires when the agent is instantiated.
6.          focus.set('sandwich bread')     #set focus buffer to direct program flow

7.      def bread_bottom(focus='sandwich bread'):  # if focus =’sandwich bread’ , fire rule
8.          print "I have a piece of bread"        #
9.          focus.set('stop')           #set focus buffer to direct program flow
```

Please note the line numbers are provided for references only. They are not part of the program. And Python is whitespace sensitive; indentations are important for defining the levels for code structures.

Central Executive Control

The central executive control of an agent is one of the core components of the ACT-R theory. The main function of the central executive control is to represent consciousness and intentionality in the agent.

When an agent is created, the central executive control function starts to operate as the agent's awareness. It continuously examines the buffers (also see Buffers section) and the module states to determine if a particular production rule is to be activated. A production rule is "fired" or activated when the conditions of the rule are completely matched. i.e. when all the buffer values are matched with the conditions stated in the condition-part of a production rule. Production rules are activated one rule at a time and each firing consumes 50 millisecond of time. For more complex models, multiple rules could be activated in parallel in different agents or modules.

An example for a production rule is defined in Example 1 line 7: the label for the rule is 'bread_bottom', the rule condition is: focus = 'sandwich bread' and the actions for the production are defined in line 8 and 9. For more details on productions rules, see Production Rules section.

From the programming perspective, the central executive control and the production rules together are responsible for the algorithm of the program/agent. In Example 1, the rules are fired or executed according to the sequenced set by the focus buffer. You can find a better demonstration of the program flow in Example 2 – follow how the focus buffer is set and see how the task flow occurs.

Declarative Memory

Declarative memory refers to memories that can be consciously recalled and discussed. The two types of declarative memories are semantic memory and episodic memory. Semantic memory is knowledge independent of time and place; a piece of information. For example, "A Robin is a bird" can be stored as a semantic memory chunk. Episodic memory is factual knowledge of personal experience in a specific time and place.

Declarative memory is subjected to forgetting. Declarative memories are best established by using active recall.

In ACT-R theory declarative memories are represented by "chunks". Chunks are used to communicate information between modules through the buffers.

From the programming perspective, declarative memories are the data store for the agent. In Python ACT-R, the agent can remember or learn new information by creating new chunks. Example 2 shows how an agent can store and retrieve information from declarative memory.

Example 2: Declarative Memory

```
1.  import ccm                          # import ccm module library for Python ACT-R classes
2.  from ccm.lib.actr import *

3.  class MyAgent(ACTR):                # Defining  an act-r agent of type MyAgent.

4.      focus=Buffer()                  # Creating the goal buffer for the agent
5.      DMbuffer=Buffer()   # create a buffer for the declarative memory
6.      DM=Memory(DMbuffer)   # create a declarative memory object called DM.
7.
8.      def init():                     # this rule fires when the agent is instantiated.
9.          DM.add('condiment:mustard')         # put a chunk into DM; has a slot condiment
10.         focus.set('get_condiment')          #set focus buffer to goto recalling rule
11.
12.     def recalling(focus='get_condiment'):
13.         print "Retrieving the condiment chunk from DM into the DM buffer"
14.         DM.request('condiment:?')     #'?' means that slot can match any content
15.         focus.set('sandwich condiment')    # set focus buffer to goto condiment rule
16.
17.     def condiment (focus='sandwich condiment', DMbuffer='condiment:?condiment'):
18.         print "My recall of the condiment from memory is...."   # DMbuffer is used as a
    condition for the rule.
19.         print condiment   # condiment  variable here is associated with ?condiment var.
20.         focus.set('stop')                   # set focus buffer to goto stop_production rule

21.     def stop_production(focus='stop'):
22.         self.stop()         #  'self' is the agent object and '.stop' is the method for stopping
```

Example 2 above is an extension of Example 1. It has included declarative memory in the model. Line 5 and 6 are instructions for creating a declarative memory object in an agent. Line 5 uses the Buffer() class to create DMbuffer, a new buffer for the memory object[5]. Line 6 uses the Memory() class and DMbuffer to create a memory object called DM for the agent.

Line 9 shows how to add a new chunk to the declarative memory. It simply uses the '.add' method of the memory object to add a chunk in DM. Note that a single quoted string is used as a parameter for the method. In general, the

[5] In Python everything is considered an object; objects are instantiated from classes.

method of the object is invoke by a "."Operator. In this case the method is ".add" – "DM.add". The quoted string represents a chunk . A chunk is a list of "slot-value" pairs. The ":" inside the string is a separator which is used to separate the slot name from its value. In line 9 ('condiment:mustard') we have a slot called 'condiment' and the value is 'mustard'.

In Python ACT-R, to retrieve a value from declarative memory for an operation usually takes two rules. One rule to make a request (line 14) to the memory object using a cue (slot name) and another rule to process the retrieved value in the memory buffer (line 17). Notice that the "?" notation in ".request" (line 14 - 'condiment:?') means the "value" of the slot can be anything as long as the chunk has a condiment slot. However, the meaning of the "?" changes in line 17, in line 17 "?" represents the definition and assignment of a variable for the value of the slot. The variable can then be used by agent for further processing. In Example 2, the variable "?condiment" has the value "mustard" assigned to it from the DMbuffer. And notice that the "?condiment" variable is being used in line 19 without the "?" – "print condiment" (the "?" of the buffer variables are dropped before these variables are used with regular Python commands; 'print' is a Python command). "mustard" will be printed to the console as a result of this instruction.

The convention of naming the buffer variables is to use the slot name. For example, slot condiment the buffer value variable is "?condiment". The general convention is: <slot name>:?<slot name>.

Memory Retrieval – Sub Symbolic Parameters

Memories do fade and some become inaccessible. The retrieval of a particular chunk is mainly dependent on the activation level of the chunk. For more complex models, there are a variety of sub-modules which can adjust the activation of the chunks in memory, giving them different probabilities of being retrieved. However, if these sub-modules have not been applied, by default, the declarative memory system will randomly choose one of the many chunks that match the request.

The different parameters that support retrieval modeling are:

- Latency: controls the relationship between activation and how long it takes to recall the chunk
- threshold: chunks must have activation greater than or equal to this to be recalled (can be set to None)
- maximum_time: no retrieval will take longer than this amount of time
- finst_size: the FINST system allows you to recall items that have not been recently recalled. This controls how many recent items are remembered

- first_time: controls how recent recalls must be for them to be included in the FINST system.

More details on applications of these parameters can be found in the following URL:

http://sites.google.com/site/pythonactr/reference-material/declarative-memory

Buffers

Buffers in ACT-R are the built in interface between modules. Each buffer is connected to a specific module or core component and has a unique name by which it is referenced. For example, the focus buffer is connected with executive control or the retrieval buffer with declarative memory. A buffer serves to relay requests for actions to its module, and it can hold one chunk which is visible to all other modules.

In response to a request, a module will usually generate one or more events to perform some action(s) and may place a chunk into the buffer to indicate the result of that action. Any module may access or modify the chunk in any buffer at any time, and typically a module will manipulate its own buffer(s) only (e.g. using focus.set()). A buffer will also respond directly to queries to determine whether or not it currently holds a chunk and for the status of how the chunk was placed into the buffer (whether it was the result of a specific request or if it was unrequested and spontaneously placed there by the module).

From a programming perspective, the chunks held in the buffers act like 'global' data which is made available to all other modules/objects.

Buffers are specifically important to the production system. Production rule conditions are defined by what's in the buffers; what is in the buffers collectively defines a mind state. And production rules are activated/fired according to the values in the buffers. Therefore, the steps in a cognitive task are always guided by the buffer values.

In summary, buffers are important for the following functions: 1. Buffers are workspace for the connected module to process information, 2. Global (sharing) data store or status registers, 3. Buffers represent a mind-state that drives the production system.

Production System (Rules)

Production system represents procedural knowledge that is well embedded in the agent. Production system consists of a set of production rules. Production rules express the notion of "If <conditions> then <actions>" procedures. The syntax of a production rule is best described by an example – see Example 2 line 21 and 22. The rule starts with the keyword "def" followed by the rule name "stop_production"; the condition to activate this rule is when focus buffer is equal to 'stop'; the action is "self.stop" stopping the agent.

Example 3: Defining a Production Rule

```
def stop_production(focus='stop'):
    self.stop()        #  'self' is the agent object and '.stop' is the method for stopping
```

Environments

Most non-trivial tasks require an agent to interact with the environment. When modeling an environment, the key question to be answered is: what elements are relevant and important to my problem? The important first step to model an environment is to identify all of the external elements that are relevant to the interactions with your agent. These required elements or objects can be considered as a part of the physical 'external' environment or a part of the 'internal' cognitive environment. Once they are identified, the external elements can be modeled as "chunk-like" objects by using the perceptual-motor (PM) module (SOSVision module) or they could be modeled by user created Python ACT-R agents/modules.

The following example shows how to define an environment with external chunk-like elements for making a sandwich.

Example 4: Defining a Sandwich Making Environment

```
1.  class Subway(ccm.Model):   # A class for the environment (env).
2.      bread=ccm.Model(isa='bread',location='on_counter') # an element (chunk-like) in the env.
3.      cheese=ccm.Model(isa='cheese',location='on_counter')
4.      ham=ccm.Model(isa='ham',location='on_counter')
5.      bread_top=ccm.Model(isa='bread_top',location='on_counter')
6.      sue_voice=ccm.Model(isa='voice',message='none')    # sounds are part of the environment
```

Line 1 in Example 4 declares an environment object class "Subway" which is inherited from ccm.Model class in CCMSuite. Line 2 to 6 declares different objects in the "Subway" environment. Note that all the objects are created from the ccm.Model class, this is how you create an object in an environment. Each object is defined like a chunk. The list of attributes consist of "slot-value" pairs, the slot name is separated from the value by an "=" sign and not a ":" as in a chunk definition.

Chapter 4:
A Sample Model

What we will discuss in this section:

- ✓ A sample model.

A Sample Model

The previous chapter discusses all the necessary basic core components of Python ACT-R that are required to build a cognitive model. In this chapter, I will walk the reader through a complete running model which uses most of the components covered.

The following example (sos-vision.py, taken from Dr. Robert West's course website) shows how the vision (SOSVision) module can be used to construct a model that interacts with an environment.

This model specifies an agent that looks for a red-stripe sock in a drawer.

Example 5: A Vision Module Model (SOSVision)

```
1.  import ccm
2.  from ccm.lib.actr import *
3.  log=ccm.log(html=True)
4.  ####### Section1:  Class definitions ######################
5.  class Sock_drawer(ccm.Model):
6.       sock1=ccm.Model(isa='sock',location='in_drawer',feature1='red_stripe',salience=0.5)
7.       sock2=ccm.Model(isa='sock',location='in_drawer',feature1='blue_stripe',salience=0.5)
8.  class MyAgent(ACTR):
9.     focus_buffer=Buffer()
10.    visual_buffer=Buffer()
11.    vision_module=SOSVision(visual_buffer,delay=0)
12. def init():
13.    focus_buffer.set('look')
14. def find(focus_buffer='look'):
15.    vision_module.request('isa:sock location:in_drawer')
16.    focus_buffer.set('get_sock')
17.    print "I am looking for a sock"
18. def found(focus_buffer='get_sock',visual_buffer='isa:sock location:in_drawer feature1:?feature1'):
19.    print('I found a sock')
20.    focus_buffer.set('check ?feature1')
21.    visual_buffer.clear
22. def check_yes(focus_buffer='check red_stripe'):
23.   focus_buffer.set('stop')
24.   visual_buffer.clear
25.   print('it has a red stripe')
26. def check_no(focus_buffer='check blue_stripe'):
27.  focus_buffer.set('look')
28.  visual_buffer.clear
29.  print('it has a blue stripe')
30. def not_found(focus_buffer='get_sock',visual_buffer=None):
31.  focus_buffer.set('look')
32.  visual_buffer.clear
33.  print('where is that sock?')
34. def fin (focus_buffer='stop'):
35.   self.stop()
36. ####### Section 2:Model starts running from here################
37. tim=MyAgent()
38. env=Sock_drawer()
39. env.agent=tim
40. ccm.log_everything(env)
41. env.run()
42. ccm.finished()
```

Lines 1 and 2 are import statements for including the 'ccm' module for the model. The 'ccm' module provides all the necessary Python ACT-R classes such as "ccm.Model" and "ACTR" for modeling.

Line 3 (log=ccm.log(html=True) turns on the HTML formatted trace output. Figure 4.1 shows a sample of the output.

time	agent				
	focus_buffer	production	vision_module		visual_buffer
	chunk		busy	error	chunk
0.000	look	find	False	False	
0.050	get_sock	found	False	False	*feature1:*red_stripe *isa:*sock *location:*in_drawer *salience:*0.5
0.100	check red_stripe	check_yes			
0.150	stop				

agent.code_buffer.chunk	None
agent.production_match_delay	0
agent.production_threshold	None
agent.production_time	0.05
agent.production_time_sd	None
agent.vision_module.delay	0
agent.vision_module.delay_sd	None
sock1.feature1	red_stripe
sock1.isa	sock
sock1.location	in_drawer
sock1.salience	0.5
sock2.feature1	blue_stripe
sock2.isa	sock
sock2.location	in_drawer
sock2.salience	0.5

Figure 4.1 HTML Log Trace

Lines 5 to 7 specify a sock_drawer class for the creation of an environment object. The sock_drawer class is inherited from the ccm.Model class from ccm library. There are two sock objects in sock_drawer: sock1 ans sock2. Each sock object is defined with four attributes (slot names): "isa", "location", "feature1", and "salience". The salience setting on the object determines how fast it is notice by the vision module.

Lines 8 to 35 specify a MyAgent class for the creation of an agent object. The MyAgent class is inherited from the ACTR class from ccm library. The focus buffer and the visual buffers are specified in line 9 and 10.

The visual module is specified in line 11. The parameter "delay" is set to 0 means the results of the visual search are placed in the visual buffer right after the

request. A request takes 50 msec and retrieval takes 50 msec. So it will take 100 msec to get the results at minimum.

Line 12 and 13 specify the initialization of the agent, which sets the focus_buffer to 'look'. And the setting of focus buffer to 'look' will cause the 'find' production rule to fire. Notice the same ".set" mechanism is used to set the values of the buffers and consequently indirectly guiding the agent's task. When the production rule 'find' is activated, it makes a request (line 15) to the visual module to obtain what the module has seen in terms of slot-values "isa:sock" and "location:in_drawer". A chunk that matches these conditions will be put in the visual_buffer. Production 'find' has set the conditions for activating production 'found' (line 18). Notice feature1:?feature1' means assigning feature1 value in the visual buffer to variable "?feature1", which is being used to set up the condition for firing the next rule.

Line 20, it depends on what value "?feature1" has, either the focus buffer will be set to "check red_stripe" or "check blue_stripe" and that will cause the appropriate rule to fire next. If production "check_yes" is fired, that means the focus buffer is set to "check red_stripe" and that means the red-stripe sock is found, the agent will fire production 'fin' and stop. Otherwise, it will fire production 'look' again.

The actual running of the model starts in line 37 when the agent "tim" is instantiated or created from the MyAgent class definition. (tim = MyAgent()). And in line 38 the environment object is instatiated. In line 39 the agent is assigned as an object in the environment – this is how the object buffers are shared. Line 40 is a trace instruction to log all output from the environment object (env). Line 41 initiates the running of the model "env". When "env" stops, instruction line 42 will cleanup all ccm objects.

The following is the console output of this model:

```
0.000 sock1.feature1 red_stripe

0.000 sock1.salience 0.5

0.000 sock1.location in_drawer

0.000 sock1.isa sock

0.000 agent.production_threshold None

0.000 agent.production_time_sd None

0.000 agent.production_match_delay 0

0.000 agent.production_time 0.05

0.000 agent.code_buffer.chunk None

0.000 agent.vision_module.busy False

0.000 agent.vision_module.delay 0
```

0.000 agent.vision_module.delay_sd None

0.000 agent.vision_module.error False

0.000 agent.focus_buffer.chunk None

0.000 agent.visual_buffer.chunk None

0.000 sock2.feature1 blue_stripe

0.000 sock2.salience 0.5

0.000 sock2.location in_drawer

0.000 sock2.isa sock

0.000 agent.focus_buffer.chunk look

0.000 agent.production find

0.050 agent.production None

0.050 agent.focus_buffer.chunk get_sock

I am looking for a sock

0.050 agent.vision_module.error False

0.050 agent.vision_module.busy True

0.050 agent.vision_module.busy False

0.050 agent.visual_buffer.chunk feature1:blue_stripe isa:sock location:in_drawer salience:0.5

0.050 agent.production found

0.100 agent.production None

I found a sock

0.100 agent.focus_buffer.chunk check blue_stripe

0.100 agent.production check_no

0.150 agent.production None

0.150 agent.focus_buffer.chunk look

it has a blue stripe

0.150 agent.production find

0.200 agent.production None

0.200 agent.focus_buffer.chunk get_sock

I am looking for a sock

0.200 agent.vision_module.error False

0.200 agent.vision_module.busy True

0.200 agent.vision_module.busy False

0.200 agent.visual_buffer.chunk feature1:red_stripe isa:sock location:in_drawer salience:0.5

0.200 agent.production found

0.250 agent.production None

I found a sock

0.250 agent.focus_buffer.chunk check red_stripe

0.250 agent.production check_yes

0.300 agent.production None

0.300 agent.focus_buffer.chunk stop

it has a red stripe

0.300 agent.production fin

0.350 agent.production None

Chapter 5: Start Your Own Model

What we will discuss in this section:

- ✓ Modeling steps
- ✓ Useful programming practices
- ✓ Epilogue

Modeling Steps

Here is an outline that you might want to follow to build your own model:

1. Define the cognitive task that you want to investigate.
2. Develop your theory. Your theory is the explanation of how the cognitive task that you are investigating could occur. Your explanations and your model is based on the pACT-R framework and ACT-R core components.
3. Identify the required objects in the environment for your tasks and agent(s) (define the required attributes of these objects for your task and agents).
4. Define your agent(s):
    - Define your declarative memory chunks – their slots and values
    - Define your production rules – their conditions and actions.
    - Optionally define new modules or agents to support the actions of your theory (your production rules).
    - Consider sub symbolic parameters requirements.

Useful Programming Practices

To avoid possible frustrations, here are some programming suggestions that may prove useful:

1. Start with a basic working template.

2. Always introduce code in small units and test them immediately and incrementally.
3. After the new code is tested, save the model as often as possible to files as new versions for backup and roll-back purposes.
4. Comment your code.
5. Use '#' to comment out test code.
6. Always start with code that has been tested before.
7. For debugging purposes:
    - use print statements,
    - use console and HTML trace logs,
    - use "#" to temporarily comment out suspect codes, and
    - Set focus buffer to jump to other test codes.

Epilogue

The design of this user guide is to provide new pACT-R users an introductory tutorial. Python ACT-R is a rich and flexible framework for cognitive modeling. There are many areas that have not been adequately or appropriately addressed in this guide. However, I hope by reading through this tutorial, the users will now feel more equipped to explore many other interesting topics in Python ACT-R. Happy modeling!