

# AdaCoder: Adaptive Prompt Compression for Programmatic Visual Question Answering (Supplementary material)

Anonymous Author(s)

## A. COMPRESSED PROMPTS

As described in Sec. 3.2, the compression phase of AdaCoder creates a set of compressed prompts. This section provides the resulting compressed prompts.

### A.1 Compressed code snippets

The compressed code snippets are generated by

$$\hat{c}_t = \pi(p_{\text{pre}} + r_{\text{code}} + r_{\text{sp}}(d_t)), \quad (1)$$

where  $p_{\text{pre}}$  is the original preprompt,  $r_{\text{code}}$  is the instruction to write code snippets,  $r_{\text{sp}}$  is an additional instruction to write code specialized for a specific question type with a placeholder to insert the definition of question type  $d_t$ , and  $t \in Y = \{\text{obj}, \text{cat}, \text{rel}, \text{attr}, \text{global}\}$  is one of the five question types. The five compressed code snippets are shown in Figures 8-12. As shown, code snippets are rewritten, shortened, and specialized for each question type.

#### Compressed code snippets $\hat{c}_{\text{obj}}$

```
Example 1:
Question: "Is there a red car in the image?"
Python code:
...python
def execute_command(image):
    image_patch = ImagePatch(image)
    return bool_to_ynsno(image_patch.exists("red car"))
...

Example 2:
Question: "Are there any cats in the image?"
Python code:
...python
def execute_command(image):
    image_patch = ImagePatch(image)
    return bool_to_ynsno(image_patch.exists("cat"))
...

Example 3:
Question: "Is there a person standing in the image?"
Python code:
...python
def execute_command(image):
    image_patch = ImagePatch(image)
    return bool_to_ynsno(image_patch.exists("person"))
...
```

Note: In all the examples above, the `exists` method of the `ImagePatch` class is used to check if the specified object exists in the image. The `bool_to_ynsno` function is then used to convert the boolean result to "yes" or "no" for the answer.

Figure 8: Compressed code snippets for question type of 'obj'.

#### Compressed code snippets $\hat{c}_{\text{cat}}$

```
Example 1:
Question: "Are there any red apples in the image?"
Python code:
...python
def execute_command(image):
    image_patch = ImagePatch(image)
    apple_patches = image_patch.find("apple")
    red_apples = [apple for apple in apple_patches if apple.verify_property("apple", "red")]
    return bool_to_ynsno(len(red_apples) > 0)
...

Example 2:
Question: "What is the color of the car closest to the building?"
Python code:
```

```

117 """python
118 def execute_command(image):
119     image_patch = ImagePatch(image)
120     car_patches = image_patch.find("car")
121     building_patches = image_patch.find("building")
122     closest_car = min(car_patches, key=lambda car: distance(car, building_patches[0]))
123     return closest_car.simple_query("What is the color?")
124 """
125
126 Example 3:
127 Question: "Which bird is sitting on the highest branch?"
128 Python code:
129 """python
130 def execute_command(image):
131     image_patch = ImagePatch(image)
132     bird_patches = image_patch.find("bird")
133     bird_patches.sort(key=lambda bird: bird.upper)
134     highest_bird = bird_patches[-1]
135     return highest_bird.simple_query("What is the bird?")
136 """

```

Figure 9: Compressed code snippets for question type of 'cat'.

Compressed code snippets  $\hat{c}_{rel}$ 

```

136 Example 1:
137 Question: "What color is the car?"
138 Python code:
139 """python
140 def execute_command(image):
141     image_patch = ImagePatch(image)
142     car_patches = image_patch.find("car")
143     car_patch = car_patches[0]
144     return car_patch.simple_query("What is the color?")
145 """
146
147 Example 2:
148 Question: "Is the cat sitting on the mat?"
149 Python code:
150 """python
151 def execute_command(image):
152     image_patch = ImagePatch(image)
153     cat_patches = image_patch.find("cat")
154     cat_patch = cat_patches[0]
155     mat_patches = image_patch.find("mat")
156     mat_patch = mat_patches[0]
157     return bool_to_yesno(cat_patch.overlaps_with(mat_patch.left, mat_patch.lower, mat_patch.right, mat_patch.upper))
158 """
159
160 Example 3:
161 Question: "What is the person in the image holding?"
162 Python code:
163 """python
164 def execute_command(image):
165     image_patch = ImagePatch(image)
166     person_patches = image_patch.find("person")
167     person_patch = person_patches[0]
168     return person_patch.simple_query("What is the person holding?")
169 """

```

Figure 10: Compressed code snippets for question type of 'rel'.

Compressed code snippets  $\hat{c}_{attr}$ 

```

164 Example 1:
165 Question: What is the color of the foo?
166 Python code:
167 """python
168 def execute_command(image):
169     image_patch = ImagePatch(image)
170     foo_patches = image_patch.find("foo")
171     foo_patch = foo_patches[0]
172     return foo_patch.simple_query("What is the color?")
173 """
174
175 Example 2:
176 Question: On which side of the image is the bar located?
177 Python code:

```

```

233 """python
234 def execute_command(image):
235     image_patch = ImagePatch(image)
236     bar_patches = image_patch.find("bar")
237     bar_patch = bar_patches[0]
238     if bar_patch.left < image_patch.width / 2:
239         return "left"
240     else:
241         return "right"
242 """

```

Example 3:  
Question: Is the qux above the baz?  
Python code:

```

242 """python
243 def execute_command(image):
244     image_patch = ImagePatch(image)
245     qux_patches = image_patch.find("qux")
246     baz_patches = image_patch.find("baz")
247     qux_patch = qux_patches[0]
248     baz_patch = baz_patches[0]
249     if qux_patch.upper < baz_patch.lower:
250         return "yes"
251     else:
252         return "no"
253 """

```

Figure 11: Compressed code snippets for question type of 'attr'.

#### Compressed code snippets $\hat{c}_{\text{global}}$

Example 1:  
Question: Is there a red car in the image?  
Python code:

```

258 """python
259 def execute_command(image):
260     image_patch = ImagePatch(image)
261     car_patches = image_patch.find("car")
262     red_car_patches = [car_patch for car_patch in car_patches if car_patch.verify_property("car", "red")]
263     return bool_to_ynsno(len(red_car_patches) > 0)
264 """

```

Example 2:  
Question: What is the color of the largest object in the image?  
Python code:

```

265 """python
266 def execute_command(image):
267     image_patch = ImagePatch(image)
268     object_patches = image_patch.find("object")
269     largest_object_patch = max(object_patches, key=lambda x: x.width * x.height)
270     return largest_object_patch.simple_query("What is the color?")
271 """

```

Example 3:  
Question: How many people are in the image?  
Python code:

```

272 """python
273 def execute_command(image):
274     image_patch = ImagePatch(image)
275     person_patches = image_patch.find("person")
276     return str(len(person_patches))
277 """

```

Figure 12: Compressed code snippets for question type of 'global'.

## A.2 Compressed definitions

The compressed API definitions are generated by

$$\hat{p}_{\text{def}} = \pi(p_{\text{pre}} + r_{\text{pre}}) \quad (2)$$

where  $p_{\text{pre}}$  is the original preprompt, and  $r_{\text{pre}}$  is the instruction to rewrite API definitions. Figure 13 shows the compressed API definitions. As shown, the API usage is summarized in short sentences.

Compressed API definitions  $\hat{p}_{def}$ 

API Reference:

Class: ImagePatch

Attributes:

- cropped\_image: array\_like - An array-like of the cropped image taken from the original image.
- left: int - The position of the left border of the crop's bounding box in the original image.
- lower: int - The position of the lower border of the crop's bounding box in the original image.
- right: int - The position of the right border of the crop's bounding box in the original image.
- upper: int - The position of the upper border of the crop's bounding box in the original image.
- width: int - The width of the cropped image.
- height: int - The height of the cropped image.
- horizontal\_center: float - The horizontal center of the cropped image.
- vertical\_center: float - The vertical center of the cropped image.

Methods:

- find(object\_name: str) -> List[ImagePatch]: Returns a list of ImagePatch objects matching object\_name contained in the crop.
- exists(object\_name: str) -> bool: Returns True if the object specified by object\_name is found in the image, and False otherwise.
- verify\_property(object\_name: str, visual\_property: str) -> bool: Returns True if the object possesses the visual property, and False otherwise.
- best\_text\_match(option\_list: List[str]) -> str: Returns the string that best matches the image.
- simple\_query(question: str = None) -> str: Returns the answer to a basic question asked about the image.
- compute\_depth() -> float: Returns the median depth of the image crop.
- crop(left: int, lower: int, right: int, upper: int) -> ImagePatch: Returns a new ImagePatch cropped from the current ImagePatch.
- overlaps\_with(left: int, lower: int, right: int, upper: int) -> bool: Returns True if a crop with the given coordinates overlaps with this one, else False.
- llm\_query(question: str, long\_answer: bool = True) -> str: Answers a text question using GPT-3.

Functions:

- best\_image\_match(list\_patches: List[ImagePatch], content: List[str], return\_index=False) -> Union[ImagePatch, int]: Returns the patch most likely to contain the content.
- distance(patch\_a: ImagePatch, patch\_b: ImagePatch) -> float: Returns the distance between the edges of two ImagePatches.
- bool\_to\_yesno(bool\_answer: bool) -> str: Convert a boolean value to "yes" or "no".
- coerce\_to\_numeric(string) -> float: Returns a float after removing any non-numeric characters from the input string.

Figure 13: Compressed API definitions. AdaCoder reduced the token length by 72.5%.

## B. ORIGINAL PREPROMPT AND IMAGE PROCESSING APIS

This section describes the image processing APIs used in our experiments, as well as the original ViperGPT prompt. As described in Section 4.1, the API set consists of basic image and text processing functions. Specifically, it consists of the ImagePatch class and a set of auxiliary functions. The original preprompt  $p_{def}$  is given as the concatenation of prompts in Figures 14-28.

### B.1 Class and methods

The ImagePatch class is a class to store a image region. There are several code examples for each method, resulting in 13 examples. Below we describe the definition of ten methods for the ImagePatch class.

- 0) \_\_init\_\_ method.** This method is the initialization method for the ImagePatch class. It takes as input an image and the position of the bounding box. Figure 14 shows the first section of the class definition, including the definition of the \_\_init\_\_ method.
- 1) find method.** This method returns a list of ImagePatch objects matching the given object name. Figure 15 shows the method definition and an example code that uses this method.
- 2) exists method.** This method returns True if and only if the specified object is found in the image. Figure 16 shows the method definition and an example code that uses this method.
- 3) verify\_property method.** This method returns True if and only if the object possesses the visual property. Figure 17 shows the method definition and an example code that uses this method.
- 4) best\_text\_match method.** This method returns the string that best matches the image. Figure 18 shows the method definition and an example code that uses this method.
- 5) simple\_query.** This method answers to a basic question asked about the image. Figure 19 shows the method definition and three examples of code that uses this method.
- 6) compute\_depth.** This method returns the median depth of the image crop. Figure 20 shows the method definition and an example code that uses this method.
- 7) crop method.** This method returns a new image crop. Figure 21 shows the method definition and an example code that uses this method.
- 8) overlaps\_with.** This method returns True if and only if a crop with the given coordinates overlaps. Figure 22 shows the method definition and an example code that uses this method.
- 9) llm\_query.** This method calls an LLM to answer a text question. Figure 23 shows the method definition and three examples of code that uses this method.

First section of original API definitions  $p_{def}$ 

```
import math

class ImagePatch:
    """A Python class containing a crop of an image centered around a particular object, as well as relevant information.
```

```

465 """
466 Attributes
467 -----
468 cropped_image : array_like
469     An array-like of the cropped image taken from the original image.
470 left, lower, right, upper : int
471     An int describing the position of the (left/lower/right/upper) border of the crop's bounding box in the original image.
472
473 Methods
474 -----
475 find(object_name: str)->List[ImagePatch]
476     Returns a list of new ImagePatch objects containing crops of the image centered around any objects found in the
477     image matching the object_name.
478 exists(object_name: str)->bool
479     Returns True if the object specified by object_name is found in the image, and False otherwise.
480 verify_property(property: str)->bool
481     Returns True if the property is met, and False otherwise.
482 best_text_match(option_list: List[str], prefix: str)->str
483     Returns the string that best matches the image.
484 simple_query(question: str=None)->str
485     Returns the answer to a basic question asked about the image. If no question is provided, returns the answer to "What is this?".
486 llm_query(question: str, long_answer: bool)->str
487     References a large language model (e.g., GPT) to produce a response to the given question. Default is short-form answers, can be made
488     long-form responses with the long_answer flag.
489 compute_depth()->float
490     Returns the median depth of the image crop.
491 crop(left: int, lower: int, right: int, upper: int)->ImagePatch
492     Returns a new ImagePatch object containing a crop of the image at the given coordinates.
493
494 """
495 def __init__(self, image, left: int = None, lower: int = None, right: int = None, upper: int = None):
496     """Initializes an ImagePatch object by cropping the image at the given coordinates and stores the coordinates as
497     attributes. If no coordinates are provided, the image is left unmodified, and the coordinates are set to the
498     dimensions of the image.
499     Parameters
500     -----
501     image : array_like
502         An array-like of the original image.
503     left, lower, right, upper : int
504         An int describing the position of the (left/lower/right/upper) border of the crop's bounding box in the original image.
505
506     """
507     if left is None and right is None and upper is None and lower is None:
508         self.cropped_image = image
509         self.left = 0
510         self.lower = 0
511         self.right = image.shape[2] # width
512         self.upper = image.shape[1] # height
513     else:
514         self.cropped_image = image[:, lower:upper, left:right]
515         self.left = left
516         self.upper = upper
517         self.right = right
518         self.lower = lower
519
520     self.width = self.cropped_image.shape[2]
521     self.height = self.cropped_image.shape[1]
522
523     self.horizontal_center = (self.left + self.right) / 2
524     self.vertical_center = (self.lower + self.upper) / 2

```

Figure 14: Definition of `__init__` method.Section of find method in `ImagePatch`

```

506 def find(self, object_name: str) -> List[ImagePatch]:
507     """Returns a list of ImagePatch objects matching object_name contained in the crop if any are found.
508     Otherwise, returns an empty list.
509     Parameters
510     -----
511     object_name : str
512         the name of the object to be found
513
514     Returns
515     -----
516     List[ImagePatch]
517         a list of ImagePatch objects matching object_name contained in the crop
518
519     Examples
520     -----
521     >>> # return the foo
522     >>> def execute_command(image) -> List[ImagePatch]:
523     >>>     image_patch = ImagePatch(image)
524     >>>     foo_patches = image_patch.find("foo")
525     >>>     return foo_patches
526     """
527     return find_in_image(self.cropped_image, object_name)

```

Figure 15: Definition of find method.

Section of exists method in  $\hat{p}_{def}$ 

```

def exists(self, object_name: str) -> bool:
    """Returns True if the object specified by object_name is found in the image, and False otherwise.
    Parameters
    -----
    object_name : str
        A string describing the name of the object to be found in the image.

    Examples
    -----
    >>> # Are there both foos and garply bars in the photo?
    >>> def execute_command(image)->str:
    >>>     image_patch = ImagePatch(image)
    >>>     is_foo = image_patch.exists("foo")
    >>>     is_garply_bar = image_patch.exists("garply bar")
    >>>     return bool_to_yesno(is_foo and is_garply_bar)
    """
    return len(self.find(object_name)) > 0

```

Figure 16: Definition of exists method.

Section of verify\_property method in  $\hat{p}_{def}$ 

```

def verify_property(self, object_name: str, visual_property: str) -> bool:
    """Returns True if the object possesses the visual property, and False otherwise.
    Differs from 'exists' in that it presupposes the existence of the object specified by object_name, instead checking whether the object
    possesses the property.
    Parameters
    -----
    object_name : str
        A string describing the name of the object to be found in the image.
    visual_property : str
        A string describing the simple visual property (e.g., color, shape, material) to be checked.

    Examples
    -----
    >>> # Do the letters have blue color?
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     letters_patches = image_patch.find("letters")
    >>>     # Question assumes only one letter patch
    >>>     return bool_to_yesno(letters_patches[0].verify_property("letters", "blue"))
    """
    return verify_property(self.cropped_image, object_name, property)

```

Figure 17: Definition of verify\_property method.

Section of best\_text\_match method in  $\hat{p}_{def}$ 

```

def best_text_match(self, option_list: List[str]) -> str:
    """Returns the string that best matches the image.
    Parameters
    -----
    option_list : str
        A list with the names of the different options
    prefix : str
        A string with the prefixes to append to the options

    Examples
    -----
    >>> # Is the foo gold or white?
    >>> def execute_command(image)->str:
    >>>     image_patch = ImagePatch(image)
    >>>     foo_patches = image_patch.find("foo")
    >>>     # Question assumes one foo patch
    >>>     return foo_patches[0].best_text_match(["gold", "white"])
    """
    return best_text_match(self.cropped_image, option_list)

```

Figure 18: Definition of best\_text\_match method.

Section of simple\_query method in  $\hat{p}_{def}$ 

```

def simple_query(self, question: str = None) -> str:
    """Returns the answer to a basic question asked about the image. If no question is provided, returns the answer
    to "What is this?". The questions are about basic perception, and are not meant to be used for complex reasoning
    or external knowledge.
    Parameters
    -----
    question : str
        A string describing the question to be asked.

    Examples
    -----

```

```

697 """
698 >>> # Which kind of baz is not fredding?
699 >>> def execute_command(image) -> str:
700 >>>     image_patch = ImagePatch(image)
701 >>>     baz_patches = image_patch.find("baz")
702 >>>     for baz_patch in baz_patches:
703 >>>         if not baz_patch.verify_property("baz", "fredding"):
704 >>>             return baz_patch.simple_query("What is this baz?")
705
706 >>> # What color is the foo?
707 >>> def execute_command(image) -> str:
708 >>>     image_patch = ImagePatch(image)
709 >>>     foo_patches = image_patch.find("foo")
710 >>>     foo_patch = foo_patches[0]
711 >>>     return foo_patch.simple_query("What is the color?")
712
713 >>> # Is the second bar from the left quuxy?
714 >>> def execute_command(image) -> str:
715 >>>     image_patch = ImagePatch(image)
716 >>>     bar_patches = image_patch.find("bar")
717 >>>     bar_patches.sort(key=lambda x: x.horizontal_center)
718 >>>     bar_patch = bar_patches[1]
719 >>>     return bar_patch.simple_query("Is the bar quuxy?")
720 """
721 return simple_query(self.cropped_image, question)

```

Figure 19: Definition of simple\_query method.

Section of compute\_depth method in  $\hat{p}_{def}$ 

```

719 def compute_depth(self):
720     """Returns the median depth of the image crop
721     Parameters
722     -----
723     Returns
724     -----
725     float
726         the median depth of the image crop
727
728     Examples
729     -----
730     >>> # the bar furthest away
731     >>> def execute_command(image)->ImagePatch:
732     >>>     image_patch = ImagePatch(image)
733     >>>     bar_patches = image_patch.find("bar")
734     >>>     bar_patches.sort(key=lambda bar: bar.compute_depth())
735     >>>     return bar_patches[-1]
736     """
737     depth_map = compute_depth(self.cropped_image)
738     return depth_map.median()

```

Figure 20: Definition of compute\_depth method.

Section of crop method in  $\hat{p}_{def}$ 

```

737 def crop(self, left: int, lower: int, right: int, upper: int) -> ImagePatch:
738     """Returns a new ImagePatch cropped from the current ImagePatch.
739     Parameters
740     -----
741     left, lower, right, upper : int
742         The (left/lower/right/upper)most pixel of the cropped image.
743
744     Returns
745     -----
746     ImagePatch
747         New ImagePatch cropped from the current ImagePatch.
748
749     Examples
750     -----
751     >>> # What is in upper left of the image?
752     >>> def execute_command(image)->ImagePatch:
753     >>>     image_patch = ImagePatch(image)
754     >>>     upper_left_patch = image_patch.crop(image_patch.left, round((image_patch.upper+image_patch.lower)/2),
755     >>>         round((image_patch.left+image_patch.right)/2), image_patch.upper)
756     >>>     return upper_left_patch.simple_query("What is it?")
757     """
758     return ImagePatch(self.cropped_image, left, lower, right, upper)

```

Figure 21: Definition of crop method.

Section of overlaps\_with method in  $\hat{p}_{def}$ 

```

def overlaps_with(self, left, lower, right, upper) -> bool:
    """Returns True if a crop with the given coordinates overlaps with this one,
    else False.
    Parameters
    -----
    left, lower, right, upper : int
        the (left/lower/right/upper) border of the crop to be checked

    Returns
    -----
    bool
        True if a crop with the given coordinates overlaps with this one, else False

    Examples
    -----
    >>> # black foo on top of the qux
    >>> def execute_command(image) -> ImagePatch:
    >>>     image_patch = ImagePatch(image)
    >>>     qux_patches = image_patch.find("qux")
    >>>     qux_patch = qux_patches[0]
    >>>     foo_patches = image_patch.find("black foo")
    >>>     for foo_patch in foo_patches:
    >>>         if foo_patch.vertical_center > qux_patch.vertical_center and foo_patch.overlaps_with(qux_patch.left, qux_patch.lower,
    qux_patch.right, qux_patch.upper):
    >>>             return foo
    """
    return self.left <= right and self.right >= left and self.lower <= upper and self.upper >= lower

```

Figure 22: Definition of overlaps\_with method.

Section of llm\_query method in  $\hat{p}_{def}$ 

```

def llm_query(self, question: str, long_answer: bool = True) -> str:
    """Answers a text question using GPT-3. The input question is always a formatted string with a variable in it.

    Parameters
    -----
    question: str
        the text question to ask. Must not contain any reference to 'the image' or 'the photo', etc.
    long_answer: bool
        whether to return a short answer or a long answer. Short answers are one or at most two words, very concise.
        Long answers are longer, and may be paragraphs and explanations. Default is True (so long answer).

    Examples
    -----
    >>> # What is the city this building is in?
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     building_patches = image_patch.find("building")
    >>>     building_patch = building_patches[0]
    >>>     building_name = building_patch.simple_query("What is the name of the building?")
    >>>     return building_patch.llm_query(f"What city is {building_name} in?", long_answer=False)

    >>> # Who invented this object?
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     object_patches = image_patch.find("object")
    >>>     object_patch = object_patches[0]
    >>>     object_name = object_patch.simple_query("What is the name of the object?")
    >>>     return object_patch.llm_query(f"Who invented {object_name}?", long_answer=False)

    >>> # Explain the history behind this object.
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     object_patches = image_patch.find("object")
    >>>     object_patch = object_patches[0]
    >>>     object_name = object_patch.simple_query("What is the name of the object?")
    >>>     return object_patch.llm_query(f"What is the history behind {object_name}?", long_answer=True)
    """
    return llm_query(question, long_answer)

```

Figure 23: Definition of llm\_query method.

## B.2 Auxiliary functions

The auxiliary function set consists of the following four functions.

- 1) **distance function.** This function returns the distance between the edges of two ImagePatch instances. Figure 24 shows the function definition and an example code that uses this function.
- 2) **best\_image\_match** This function returns the patch most likely to contain the content. Figure 25 shows the function definition and an example code that uses this function.
- 3) **bool\_to\_yn** This function convert a boolean value to “yes” or “no”. Figure 26 shows the function definition and an example code that uses this function.
- 4) **coerce\_to\_numeric** This function returns a float after removing any non-numeric characters. Figure 27 shows the function definition and an example code that uses this function.

### Section of distance function in $\hat{p}_{def}$

```
def distance(patch_a: ImagePatch, patch_b: ImagePatch) -> float:
    """
    Returns the distance between the edges of two ImagePatches. If the patches overlap, it returns a negative distance
    corresponding to the negative intersection over union.

    Parameters
    -----
    patch_a : ImagePatch
    patch_b : ImagePatch

    Returns
    -----
    float
        The distance of the two patches.

    Examples
    -----
    >>> # Return the qux that is closest to the foo
    >>> def execute_command(image):
    >>>     image_patch = ImagePatch(image)
    >>>     qux_patches = image_patch.find('qux')
    >>>     foo_patches = image_patch.find('foo')
    >>>     foo_patch = foo_patches[0]
    >>>     qux_patches.sort(key=lambda x: distance(x, foo_patch))
    >>>     return qux_patches[0]
    """
    return distance(patch_a, patch_b)
```

Figure 24: Definition of distance function.

### Section of best\_image\_match function in $\hat{p}_{def}$

```
def best_image_match(list_patches: List[ImagePatch], content: List[str], return_index=False) -> Union[ImagePatch, int]:
    """Returns the patch most likely to contain the content.

    Parameters
    -----
    list_patches : List[ImagePatch]
    content : List[str]
        the object of interest
    return_index : bool
        if True, returns the index of the patch most likely to contain the object

    Returns
    -----
    int
        Patch most likely to contain the object

    Examples
    -----
    >>> # What is the blue foo doing?
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     foo_patches = image_patch.find("foo")
    >>>     foo_patch = best_image_match(foo_patches, ["blue"])
    >>>     return foo_patch.simple_query("What is the foo doing?")
    """
    return best_image_match(list_patches, content, return_index)
```

Figure 25: Definition of best\_image\_match function.

```

Section of bool_to_yesno function in  $\hat{p}_{\text{def}}$ 
def bool_to_yesno(bool_answer: bool) -> str:
    """
    Convert a boolean value to "yes" or "no"

    Parameters
    -----
    bool_answer : bool

    Returns
    -----
    str
        "yes" or "no"

    Examples
    -----
    >>> # Is the buz green?
    >>> def execute_command(image):
    >>>     image_patch = ImagePatch(image)
    >>>     buz_patches = image_patch.find('buz')
    >>>     buz_patch = best_image_match(buz_patches, ["buz"])
    >>>     return bool_to_yesno(buz_patch.verify_property("buz", "green"))
    """
    return "yes" if bool_answer else "no"

```

Figure 26: Definition of bool\_to\_yesno function.

```

Section of coerce_to_numeric function in  $\hat{p}_{\text{def}}$ 
def coerce_to_numeric(string):
    """
    This function takes a string as input and returns a float after removing any non-numeric characters.
    If the input string contains a range (e.g. "10-15"), it returns the first value in the range.
    """
    return coerce_to_numeric(string)

```

Figure 27: Definition of coerce\_to\_numeric function.

### B.3 Coding instruction

The coding instruction prompt used in both AdaCoder (ours) and ViperGPT is shown in Figure 28.

```

Section of coding instruction in  $\hat{p}_{\text{def}}$ 
Write a function using Python and the ImagePatch class (above) that could be executed to provide an answer to the query.

Consider the following guidelines:
- Use base Python (comparison, sorting) for basic logical
  operations, left/right/up/down, math, etc.
- Use the llm_query function to access external information and
  answer informational questions not concerning the image.

```

Figure 28: Coding instruction.