How To Make Your Own Text Adventure On A Computer

Schnell Style (but in Python)

If you have made a text adventure in your mind that you really like, and you're tired of dictating it and would rather make it in a computer language, this page is for you! I'm teaching you a shortcut to make a text adventure, for those who actually know Python.

1. Decide the map of your text adventure. Here are some tips:

- Feel free to introduce to the player.
- It is optional for you to ask for the player's name or not.
- There should be an indication for most things that don't make much sense.
- If you have a score, indicate whenever it changes.
- A good prompt is "<" (don't keep using "What next? "). You can also do "< ". But it would be a fallacy if you require the player to enter "<" every time.
- Unless you feel fit for saving and restoring, don't make such a long adventure. A long adventure needs saving and restoring, incase it takes a player two days to win.
- Make sure it is possible to win.

For best results when using this style:

- Don't require multiple descriptions and present-indications for most objects.
- Unless this seems impossible, don't force a description, present-indication or section description to change properties.

Take a sheet of paper. Draw a map of your game, and put '+' signs on all the sections. Number your sections from 0, the way you want them.

2. Set your properties. Anything that's not a condition of whether an object is in the inventory, or in a certain area is a separate property. Use it as a variable and remember the variable; set it to `None` if it doesn't do anything yet.

3. Initialize the sections. You will write the following things:

- A variable `current_section` that is the index of the current section
- A variable `sections` that has all of the sections' descriptions
- A dictionary `sec` which converts the short name of a section to its index
- A variable `dungeon_map` that indicates the simple ways to get to places

Set `current_section` to the current section's index (according to your map). `current_section` changes from time to time. In `sections`, you have a list of tuples. The first item of the tuple (item 0) is the description, in a triple-quoted string. (The DOS width is 80, so hit enter before any word that makes the line have 80 or more characters. My `fit` module would help you do this faster.) The second item of the tuple is the short name of the section. But make sure the rooms are in the order that you numbered them on your map, otherwise the thing won't work.

Here's an example:

```
sections = [
    (
"""You are at the front of your friend's house.  Your friend's house is to the
north, and the driveway to the southeast.  There is a mailbox here next to you.""",
    "Front Of Friend's House"), #0
    (
"""You are in your friend's house.  Stairs lead up to the west, the kitchen to the
north, and the living room to the east.  The exit is to the south.""",
    "Friend's House"), #1
#and so on
#23456789 123456789 123456789 123456789 123456789 123456789 123456789 123456789["]
]
```

Remember to put the descriptions first, and the triple-quoted strings aligned to the left. In

the description, give the directions, and any permanent object search description. Now make a `sec` variable as follows: (`"Front Of Friend's House"` will be translated to `'front-of-friends-house'`)

```
sec = {}
for indexed in enumerate(sections):
    index = indexed[0]
    long_name = indexed[1][1] # indexed[1][0] is the description
    short_name = ''
    for C in long_name:
        if C in ' /': # spaces and slashes to dashes
            short_name += '-'
        elif not C in ".'": # don't use periods and apostrophes
            short_name += C.lower() # lowercase
    sec[short_name] = index
```

That will get you what you want. If you test the `sec` variable: `{'front-of-friends-house': 0, 'friends-house': 1, ...}`

The functions will refer the front of the friend's house as 0, but this is equivalent to `sec['front-of-friends-house']` which you can use!

Now make an index for each direction. Like most games, you should accept `n` for `north` etc.

```
dirs = {'north': 0, 'n': 0, 'south': 1, 's': 1,
        'west': 2, 'w': 2, 'east': 3, 'e': 3,
        'northeast': 4, 'ne': 4, 'southeast': 5, 'se': 5,
        'northwest': 6, 'nw': 6, 'southwest': 7, 'sw': 7,
        'up': 8, 'u': 8, 'down': 9, 'd': 9,
        'in': 10, 'out': 11, 'on': 10, 'off': 11,
        'enter': 10, 'exit': 11}
```

Now each direction has an index. (Don't use 'dir'; it's a built-in function.) Make a list of lists: each list in the big list is the map of the section (they must go in order as index), and each item in the small list is the result of going in the direction (the directions must go in order as indexed, too).

If you simply just can never go that way, use -1; if the room is definite the whole game, use that room's index no.; and if the move is special, use a rather large number that's not a room index. (If your game is short, try 99; if there are 100 or more sections, try 255.)

For example, this:

```
dungeon_map = [
#     n   s   e   w  ne  nw  se  sw  up  dn  in  out
    [ 1, 15, -1, -1, -1, -1, 15, -1, -1, -1,  1, -1], #0
    [ 2,  0,  3,  4, -1, -1, -1, -1,  4, -1, -1,  0], #1
# and so on.
]
```

The '#0' indicates that you can go north (or in) to room 1, and south or southeast to the driveway (considered room 15). From #1 you can go north to the kitchen, east to the living room, west and up go to the same place, and south (or out) gets you back to the front of the house. You'd use something like 99 or 255 if the move depends (or has an extra message). (Heartfelt Nightsong doesn't do it this way.)

4. Initialize the objects.

Based on objects, you will make five things here:

- references and indices of objects ('obj')
- the short names and searches of takable objects ('tk_objs')
- the descriptions of takable objects ('tk_obj_desc')
- the searches of certain permanent objects ('perm_objs')
- the descriptions of permanent objects ('perm_obj_desc')

(Special objects need descriptions.)

First, index your objects. According to Ron Schnell, takable objects have nonnegative indices and permanent objects have negative indices. In the dictionary, use every name of the object that you recognize from the user. (For a floppy disk, you should accept 'disk' because, as a one-word name, 'disk' makes more sense than 'floppy'.)

Here's a sample dictionary:

```
obj = {
    'digging': 0, 'spade': 0, 'shovel-like': 0,
    'lamp': 1, 'lantern': 1, 'light': 1,
    # and so on for the takable objects
```

```
    'computer': -1, 'personal': -1, 'active': -1, 'pc': -1, 'pac': -1,
    'bed': -2, 'mattress': -2,
    # and so on for the permanent objects
}
```

Make a list of tuples for the objects' (in order as their index!) search indication and short names.

Here's an example:

```
tk_objs = [
    ("There is a digging spade here.", "A digging spade"),
    ("There is a shiny brass lamp nearby.", "A brass lantern"),
# and so on.
]
```

Now (in order as their index) list the objects' descriptions. Put None if there is no description (you can have a description of everything). e.g.

```
tk_obj_desc = [
    "A spade that can be used as a shovel.",
    "The lamp is hand-crafted by Geppetto.",
    # and so on
]
```

Now (for 0) put None at the beginning of a list. From -1 to the last permanent object, put None if the object never moves around (the search indication is in the section description), otherwise, put a search indication. Anything moves around (like a fed animal that walks away), it needs a search indication here.

Don't forget the first `None`! You'll be using abs to refer to this list. Call this 'perm_objs' Do the descriptions of the permanent objects (again putting None at the beginning).

```
perm_obj_desc = [
    None,
    "A personal active computer available on the table.",
    "A bed with the perfect mattress for sleep.",
    # and so on.
]
```

5. Put them together.

ow combine the section indices with the object indices and write a variable for:

- the inventory (`inventory`)
- the items available in each section (`items`)

The inventory is a list of what you have in your inventory (a list of the objects' indices). The items available in each section supply the `inventory` of each section. For example, if a spade starts in a certain room, use the list item `[obj['spade']]` for that room. Remember to list the section-inventories in order of the section-indices!

6. Write the help file. It can have a menu or just a long help file.

Note: Don't use `help`; it's a built-in function for Python's help files. Use `_help` instead, and define it for exactly one argument (as a list of arguments).

Don't forget to use the `print` delimeter; putting just a plain string on a line moves you nowhere.

7. Write normal functions, such as:

- describing the section (special description indications)
- inventory
- take, drop, look
- move around (special moves)
- function to die

Either the game could quit when you die, or you could make a section 'dead' which gives the player a chance to restart, restore a saved game, or read any such score (the author of Heartfelt-Nightsong does this).

If dying doesn't really do anything, then the player could just not worry about dying.

Make a variable `dead` to indicate whether one is dead. Set `dead` to `False`.

Here is the default method for describing the section:

Make a function `special_sections` which describes any special properties of the argument section. (Use `if`'s and `elif`'s to do this.) `special_sections` needs exactly one argument, which is the section you're trying to describe any special properties about.

Then type this:

```
visited = [] # the visited rooms; if you want a room to be already visited put it in here

def describe(section):
    "Give long if we have a negative room number"
    ## This, of course, will cause room 0 to be always explained long
    global visited
    print sections[abs(section)][1]
    if section <= 0 or not section in visited:
        print sections[abs(section)][0]
    if not section in visited:
        visited.append(section)
    ## Now, everything else:
    special_sections(abs(section))
    for i in items[abs(section)]:
        if i >= 0:
            print tk_objs[i][0]
        else:
            if isinstance(perm_objs[abs(i)], str):
                print perm_objs[abs(i)]
```

This function checks the inventory: (If you have a jar that you want to indicate its objects, add to this function.)

```
def inven(args):
    print "You currently have:"
    for i in inventory:
        print tk_objs[i][1]
```

`inven` is used, because `inventory` is the list.

FUNCTIONS TO TAKE, DROP, LOOK: (Add if there is anything special)

(You can say `"take all"` to take everything)

```
def take(args):
    if not args:
        print "You must specify an object."
    else:
        if args[0] == "all":
            first_objs = list(items[current_section])
            gotsome = False
            for i in first_objs:
                if i >= 0:
                    gotsome = True
                    print "%s:" % tk_objs[i][1],
                    takeobj(i)
            if not gotsome:
                print "Nothing to take."
        else:
            if not args[0] in obj:
                print "I don't know what that is."
            else:
                takeobj(obj[args[0]])

def takeobj(x):
    global inventory, items
    if not x in items[current_section]:
        print "I do not see that here."
    else:
        if x < 0:
            print "You cannot take that."
        else:
            print "Taken."
            items[current_section].remove(x)
            inventory.append(x)

def drop(args):
    global inventory, items
    if not args:
        print "You must specify an object."
    else:
        if not args[0] in obj:
            print "I don't know what that is."
        else:
            objnum = obj[args[0]]
            if not objnum in inventory:
                print "You don't have that."
```

```
            else:
                print "Done."
                inventory.remove(objnum)
                items[current_section].append(objnum)

def examine(args): # examine = look
    if not args:
        describe(-current_section) # long description
    else:
        if not args[0] in obj:
            print "I don't know what that is."
        else:
            objnum = obj[args[0]]
            if not objnum in inventory and not objnum in items[current_section]:
                print "I don't see that here."
            else:
                if objnum >= 0:
                    desc = tk_obj_desc[objnum]
                else:
                    desc = perm_obj_desc[abs(objnum)]
                if isinstance(desc, str):
                    print desc
                else:
                    print "I see nothing special about that."
```

Now write a function that does the special move in a certain direction. Call this `special_move`. It requires exactly one argument, which is the index of the direction you try to go in. The best thing to do is to check on every special-move (according to the dungeon map), and use an `if` or `elif` to control what happens. (Whenever you go to a new section, describe it!)

Consider the number in the map which yields a special move. Replace `SPECIALNUMBER` in the following function to match that number.

```
def move(direct):
    global current_section
    newsect = dungeon_map[current_section][direct]
    if newsect == -1:
        print "You can't go that way."
    elif newsect == SPECIALNUMBER:
        special_move(direct)
    else:
        current_section = newsect
        describe(newsect)

def north(args):
    move(0)

def south(args):
    move(1)

def east(args):
    move(2)

def west(args):
    move(3)

def northeast(args):
    move(4)

def northwest(args):
    move(5)

def southeast(args):
    move(6)

def southwest(args):
    move(7)

def up(args):
    move(8)

def down(args):
    move(9)

def _in(args):
    # 'in' is a delimeter, and cannot be used as a funcname without entailing invalid syntax
    move(10)

def out(args):
    move(11)

def go(args):
    if not args:
        print "You must specify a direction."
```

```
    else:
        if not args[0] in dirs:
            print "I don't understand where you want to go."
        else:
            move(dirs[args[0]])
```

FUNCTION TO DIE:

```
def die(args):
    global dead
    print
    if args:
        print "You are dead."
    # If there is a score, show it.
    dead = True

def _quit(args): # 'quit' is a built-in
    die([])
```

8. Write the other functions in the same manner. Be sure to write every function that an input's first word translates into. Each function needs one argument as a list of arguments. If `x` is recognized as func `X`, then `x a b c` calls `X(['a', 'b', 'c'])`

You can use your own properties during these functions. But remember to global any properties in a function that you change.

Remember: When you use these functions, they require exactly one argument. So if `x` does not take any list arguments, use `x([])` (not `x()`), and if `x` takes the argument `y`, use `x([y])`; and so on, remembering the brackets.

Don't let any Python errors occur in your functions!

Also: Put '_' at the left at built-in functions and delimeters.

9. Most games have saving and restoring; you will probably want to do this too. You should have a save/restore feature if it is easy to lose (die) unexpectedly (or the text adventure is awfully long).

You shouldn't use a `saved` variable; you should save as a file. The purpose of saving is that you can shut down the computer, turn it on again, and then restore the game again. A `saved` variable won't work; all variables will reset when the program restarts.

(Don't let an error occur when you save. Use the 'try' delimeter)

Here is how you save:

```
def save_val(varname):
    "The way to reset the variable as it was."
    return "global %s\n%s = %s\n" % (varname, varname, `eval(varname)`)

def save(args):
    if not args:
        print "You must specify a filename."
    else:
        try:
            game = open('_game_' + args[0] + '.txt', 'w')
            game.write(save_val('current_section'))
            game.write(save_val('visited'))
            game.write(save_val(inventory))
            game.write(save_val(items))
            #! Game-writer, apply this to the rest of the properties that change from time to time
            game.close()
            print "Game saved."
        except Exception:
            print "Error saving to file."
```

Here is how you restore:

```
def restore(args):
    if not args:
        print "You must specify a filename."
    else:
        try:
            exec open('_game_' + args[0] + '.txt')
            print "Game restored."
            describe(current_section)
        except Exception:
            print "Could not load restore file."
```

10. I assume you're now ready to make the verblist and run the game!

The verblist is a dictionary. Have each key of the dictionary be the first word of the input (you can use spaces if you write the way Metadunnet[5] is written, but not this way). Each value is the corresponding one-arg-list function that gets called. Remember to include every single input, not just the input that doesn't match the function.

Inclusion notes:

- "take x" should take x; "drop x" should drop x. "throw" and "toss" also make sense
- "n" should lead you north, as well as "go n". Same for rest of directions
- "r" abbreviates read, "l" look, and "x" examine (you can't use "e" because of going east)

Sample verblist:

```
verblist = {
    'take': take, 'get': take, 'pick': take, 'hold': take,
    'drop': drop, 'throw': drop, 'toss': drop,
    'look': examine, 'l': examine, 'examine': examine, 'x': examine,
    'read': examine, 'r': examine, 'describe': examine, #can't use 'd' because of going down
    inventory: inven, 'i': inven, items: inven, 'die': die, 'quit': _quit,
    'help': _help, 'save': save, 'restore': restore, 'go': go,
    'north': north, 'n': north, 'south': south, 's': south,
    'east': east, 'e': east, 'west': west, 'w': west,
    'northeast': northeast, 'ne': northeast, 'southeast': southeast, 'se': southeast,
    'northwest': northwest, 'nw': northwest, 'southwest': southwest, 'sw': southwest,
    'up': up, 'u': up, 'down': down, 'd': down, 'in': _in, 'out': out,
    'on': _in, 'off': out, 'enter': _in, 'exit': out,
    # and so on with the rest of the functions
}
```

Call the verblist `verblist`. Then type the following three context figures:

```
def execprint(x):
    line = x.split()
    for c in ',:':
        line = c.join(line).split(c) # Also, get rid of `c` that's been there first
    if line:
        if not line[0] in verblist:
            print "I don't understand that."
        else:
            func = verblist[line[0]]
            args = line[1:]
            func(args)

def run_game():
    print HEADING #Is there are heading or introduction to the game?
    describe(current_section)
    while not dead:
        reply = raw_input('>').lower().split(';') or ['']
        first = True
        for i in reply:
            if not dead:
                if not first:
                    print '>'
                execprint(i)
                first = False
    if __name__ == '__main__':
        raw_input('\n') # Without this, a DOS window will automatically hit black

if __name__ == '__main__': # Important!
    run_game()
```

If all goes well, your game will be run at the prompt! Enjoy!