The Library Search program a HTML embedded Python script intended to be modular and customizable.

Source files can be found on [GitHub](https://github.com/jhutch10/LibLocate). A live example can be found at <https://library.anderson.edu/locate>. Feel free to fork and share. While the webpage is saved as a html.twig file, it can be changed to just .html if needed.

**A Guide to the Code:**

-HTML-

<script src="https://cdn.jsdelivr.net/npm/brython@3/brython.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/brython@3/brython\_stdlib.js"></script>

There are two scripts from Brython which need to be imported to make Python scripting load. These can be loaded from a CDN or hosted on your own server ([Instructions](https://www.brython.info/static_doc/en/install.html)). There are also two CSS imports (not shown) linking back to AU stylesheets for demonstration purposes, one to bootstrap and the other to a shorter style sheet specific to this tool, please update these two links to your own resources.

<body onload="brython({debug:1})">

In the <body> tag were the program will live an onload argument needs to be passed calling Brython to the section. The {debug:1} can be omitted, if desired, as it only serves to sends errors to your web browser’s console for troubleshooting. No errors will be displayed on the page regardless of setting.

<p class="flavor">*Initial Text*</p>

<p class="instructions">*Initial Text*</p>

The <p> tags classes “flavor” and “instructions” are updated by the program as menu options are selected. More updatable classes can be added, in needed, by adding more import calls in the second Python script.

<form class="field">

-Buttons & Inputs to be covered in the next section

</form>

The <form> tag creates a block of code to hold all of our inputs (buttons and textboxes) for the tool. Forms will group inputs and automatically specify that should interact with each other so, for example, a button will automatically affect any textboxes within that form only. We’ve also given a class attribute both for any overall visual edits as well as allowing it to be called by our Python code, if so needed.

<button type="reset" style="display:inline" onclick="prompt.value = '1'" id="option1">Stacks</button>

<button type="reset" style="display:none" onclick="prompt.value = 'Back'" id="back">Back</button>

<input type="text" id="prompt" style="display:none" onkeydown="return(event.keyCode!=13);" autocomplete="off">

<button type="reset" style="display:none" id="enter">Next</button>

Our inputs within the <form> tag consists of seven buttons and one text input. Each of the buttons are given the type as “reset”. This label is important because 1) it does not reload the page and 2) it clears the textbox of any inputs, giving the illusion that there is data being submitted. The first five buttons, labeled with the ids option1 – option5 (only option1 is demonstrated here, they all look almost identical), are used to select between the five collections of the Anderson library. For your institution you may need to add or delete buttons. To add extra buttons, replace the digits in prompt.value and id with the number of the new button and replace the label between > <. It’s also important to note that these collection/option buttons are given the property display:inline, this allows us to toggle visibility later. Our back and enter buttons follow a similar scheme, note: the enter button omits sending any value to the textbox, however these two along with the textbox begin as display:none since we need them to be there when the program begins but don’t want users interacting with them until they click any of the collection buttons. <Input> creates a small text box for users to type. The autocomplete=”off” command is intended to minimize user issues with mobile devices autocorrecting call numbers into words, but can be omitted if desired. Also, it is worth drawing attention to the onkeydown command. That script is responsible for blocking users from hitting the Enter key and therefore reloading the page and restarting.

<script type="text/python" class="webworker" id="worker">

-Python web worker script

</script>

<script type="text/python">

-Main logic

</script>

These two <script> tags let Brython and the site know that we are using Python. The webworker class and id are what allow us to update our <p> tags and inputs in real time as well as do the actual processing without interfering with other running scripts, if you choose to embed this on a site rather than use as a pop-up. It is important that we have the class and id attributes so we can call it from our main script.

-Python Web Worker Script-

from browser import bind, self

@bind(self, "message")

def message(evt):

try:

inprompt = str(evt.data[0])

self.send(inprompt)

except ValueError:

self.send('Please Enter A Valid Response')

This set of code is what allows the site to respond to user inputs. It sends the text to the main script and attempts to catch invalid inputs before they hit your main code. If you need more complex data manipulation, such as the chaining of two inputs or cleaning of text, they can be added before the self.send(*variable*) call. Inputs, if you have multiple, are automatically assigned starting from the top of the page. The ValueError is included as a safety net in case something goes terribly wrong, but in all likelihood will never be triggered as the program’s variables are simple and have minimal processing done on them.

-Python Main Script-

from browser import bind, document, worker, html

from browser.session\_storage import storage

import webbrowser

ins = document.select\_one('.instructions')

flavor = document.select\_one('.flavor')

hint = document.select\_one('.hint')

inputs = document.select("input")

button = document.select\_one("#enter")

field = document.select\_one('.field')

myWorker = worker.Worker("worker")

storage['mode'] = "booksel"

letters = ['A', 'AA', 'AB', 'AC', 'AD', 'AE', 'AF', 'AG', …]

letterslow = ['a', 'aa', 'ab', 'ac', 'ad', 'ae', 'af', 'ag', …]

lettersmix1 = ['A', 'Aa', 'Ab', 'Ac', 'Ad', 'Ae', 'Af', 'Ag', …]

lettersmix2 = ['a', 'aA', 'aB', 'aC', 'aD', 'aE', 'aF', 'aG', …]

The main script begins by once again importing libraries from Brython as well as the standard Python library. Then the initial variables are set, which link our HTML classes and ids to our Python code as well as create a session (temporary) key/value to help our code know where it should be looking. While a standard variable would work in this usage the choice to use short-term local storage was intended to make data importing from the HTML code easier, if needed, as well as to allow for the code to be quickly changed to local storage if there is ever a need to remember progress by swapping the import. Our final four variables setup a global list of all of the possible combinations for 2 letter call numbers. We keep these as separate lists so they will retrieve the same i value (or spot in the list) regardless of capitalization while also not requiring to use long “or” blocks in our code later.

@bind("#enter", "click")

@bind("#back", "click")

def envalue(ev):

myWorker.send([x.value for x in inputs])

The envalue function (and the preceding bind calls) ask the Web Worker to update our input data (e.data) when either the enter or back buttons are clicked. This section of the code does not need to be updated as more input fields are added since it runs a for loop and we already import all inputs on startup.

@bind("#option1", "click")

@bind("#option2", "click")

@bind("#option3", "click")

@bind("#option4", "click")

@bind("#option5", "click")

def change(event):

button.text = "Next"

document["prompt"].style.display = "inline"

document["enter"].style.display = "inline"

document["back"].style.display = "inline"

document["option1"].style.display = "none"

document["option2"].style.display = "none"

document["option3"].style.display = "none"

document["option4"].style.display = "none"

document["option5"].style.display = "none"

myWorker.send([x.value for x in inputs])

Similar to the envalue function, the change function updates the e.data variable when any of our option buttons are clicked. However, this function also makes our visual changes, hiding the collection/option buttons and revealing the next/back buttons and textbox, as well as resets the next button’s label. It is possible to move the bind commands to the previous envalue function and add the display code to our later if statements, if you so choose, but that would add an additional 36 lines of code. You may notice that the reversing display code is included in our later if statements. This is because the enter and back button may be used multiple times before needing to change the display status of our buttons. If you need to add more option buttons, copy and paste an @bind line and document line and replace the digit with that of your new button. You will also need to do that for each of the inline changes in the code, unless you want the button to only be shown on startup.

*Note:* The following section may use an older version of the tool as an example. The same principles will still apply. Some elements of the code have been omitted or abbreviated (…) for space reasons.

@bind(myWorker, "message")

def onmessage(e):

if storage['mode'] == "booksel":

if e.data == "1":

storage['mode'] = "callnum1"

flavor.text = "Search by Call Number - Stacks"

ins.text = "Please Enter the First Set of Letter(s) of the Call Number"

hint.text = "Example: Z in Z710.M23 2005 or PZ in PZ8.3.S855 LI 1995"

if e.data == "4":

webbrowser.open('images/search/Curriculum.jpg', autoraise=True)

flavor.text = "Located Behind Reference & Juvenile (Curriculum)"

ins.text = "Search Another Collection?"

hint.text = "If no graphic appeared, you may need to allow pop-ups."

document["back"].style.display = "none"

document["prompt"].style.display = "none"

document["enter"].style.display = "none"

document["option1"].style.display = "inline"

document["option2"].style.display = "inline"

document["option3"].style.display = "inline"

document["option4"].style.display = "inline"

document["option5"].style.display = "inline"

if e.data == "5":

storage['mode'] = "periodicals1"

flavor.text = "Search for Periodicals"

ins.text = "Please Enter the First Two Letters of the Periodical's Name"

hint.text = "Example: FO for Fortune or SC for Scientific American"

button.text = "Locate"

if storage['mode'] == "callnum1":

if e.data == "Back" or e.data == "back" or e.data == "BACK":

storage['mode'] = "booksel"

flavor.text = "Welcome to the Robert A. Nicholson Library"

ins.text = "Search a Different Collection?"

document["back"].style.display = "none"

document["prompt"].style.display = "none"

document["enter"].style.display = "none"

document["option1"].style.display = "inline"…

hint.text = "Stacks contains most of the library's books. If unsure, start there."

for i in range(len(letters)):

if letters[i] == e.data or letterslow[i] == e.data or lettersmix1[i] == e.data or lettersmix2[i] == e.data:

storage['mode'] = "callnum2"

storage['letter'] = str(i)

flavor.text = f'Call Number: {e.data}'

ins.text = "Please Enter the Digits Preceding the First '.'"

hint.text = "Example: 710 in Z710.M23 2005 or 8 in PZ8.3.S855 LI 1995"

button.text = "Locate"

if storage['mode'] == "callnum2":

if e.data == "Back" or e.data == "back" or e.data == "BACK":

storage['mode'] = "callnum1"

flavor.text = "Search by Call Number - Stacks"

ins.text = "Please Enter the First Set of Letter(s) of the Call Number"

hint.text = "Example: Z in Z710.M23 2005 or PZ in PZ8.3.S855 LI 1995"

for i in range(1, 10000):

if int(e.data) == i:

storage['mode'] = "booksel"

ins.text = "Search Another Collection?"

hint.text = "If no graphic appeared, you may need to allow pop-ups."

document["back"].style.display = "none"

document["prompt"].style.display = "none"

document["enter"].style.display = "none"

document["option1"].style.display = "inline"…

if int(storage['letter']) <= 26:

webbrowser.open('images/search/001a.jpg', autoraise=True)

flavor.text = letters[int(storage['letter'])] + f"{e.data} is on Shelf 001a"

if int(storage['letter']) == 27:

if int(e.data) <= 59:

webbrowser.open…

flavor.text…

if 60 <= int(e.data) <= 819:

webbrowser.open('images/search/002a.jpg', autoraise=True)

flavor.text = letters[int(storage['letter'])] + f"{e.data} is on Shelf 002a"

if int(e.data) >= 2422:

webbrowser.open…

flavor.text …

To quickly summarize, the onmessage function receives the updated input from the web worker and then decides on what to provide the end user based on the mode it is in. The following will break down one of the loops of this function, Stacks, piece by piece.

The grey block sets up our function and allows it to receive inputs. It’s important to note that every single bit of logic that refers to the actual searching part is contained below in this function. This does mean it will get very long.

In the green block, which exemplifies the basic building blocks of this Python script in a more visible manner, the code will check the local storage key “mode” and then find the corresponding if statement. Then, in response to specific alphanumeric inputs, the mode setting will be updated, the text displayed in three <p> tags will be changed, and some visual changes may occur. The more specifically, it looks to see which option button is selected and then sets up the next step. For options 1-3 there is a two-step process for retrieving the correct shelf so we set the mode to the first step (in this instance the first step for searching AU’s Stacks collection is called callnum1) as well as change our flavor, instruction, and hint texts to reflect that the user is searching stacks and that they will be entering the subject letters. Option 4 corresponds to a collection that only exists on one shelving unit therefore it simply opens our information graphic for the curriculum collection, updates the flavor text to give a descriptor of the location, provides instructions on searching for other books, changes the hint to let the user know what to do if there is an issue, and finally re-adds our option buttons (since they hide every time they are clicked). Option 5 is similar to options 1-3, however since it corresponds to a one-step search we set the next button to say “Locate” (this is completely optional, but it helps users understand where they are in the process and makes this look more impressive). If you add more option buttons you will need to add additional if e.data == statements. It is possible to make multiple buttons go to the same search by either replacing the digit in the prompt.value line earlier or adding an or to any if statement (ex: if e.data == “1” or e.data == “2”).

In the orange block, the code checks the user input against the four lists of letters that potentially make up the beginning of a call number. If it finds a match the code then saves where on the list it found that match in a session key as well as updating the three texts, button label, and mode. For example, if the user inputs “Ax” it will pass the check since Ax is in the list called “letterslow” therefore its position (24) is saved to the key “letter”, the mode is changed to callnum2, the flavor text is updated to show the user’s search so far “Call Number: Ax” (the f’…{e.data}’ allows us to not have to manually write out all of the possible lines), the instruction text asks for the following number, the hint gives an example of what the program is looking for, and the next button changes to say locate since we are proceeding to step two. If the user inputs anything besides one or two letters the textbox will clear and the program will do nothing. Option 5 skips this orange block and instead jumps to the blue block, though it still uses the letters search in place of numbers.

In both the orange and blue block there is a back option, available by either clicking back or typing any of the three styles of back and clicking the next button. These reset to tool to the previous step. The back for step one of options 1-3 is similar to the changes made in the second step and the back in step two of options 1-3 (and step one of option 5) is similar to the changes made leaving the main menu. Option 4 does not have a back button since it never leaves the main menu.

In the blue block, which exemplifies the most complex part of the program’s nested if statements, the code checks if the number input by the user is between 1 and 9,999 (this is because 1] those are the limits of what a call number can be and 2] so we can anticipate received values. If your library’s classification system allows for higher than 9999 then replace with that plus one). If the number does not match this criterion the textbox will clear and nothing will happen. Then, after that test passes, the code updates the ‘mode’ key as well as the instruction and hint texts, changes the visibility of our buttons, but does not touch the instruction text yet. Next, the code begins to search through the third layer of if statements using the key “letter” saved in the orange block (regardless of capitalization this check, and later call, will always use the “letters” list so make sure the four lists maintain the same length). When the code gets to the spot on the list where the user’s subject letter(s) are (these are the ‘if int(storage[‘letter’]) ==’ lines) it then checks if that subject is broken across multiple shelves. If so, then the program searches a fourth layer of if statements to find the specific range that corresponds to the 1-9999 number the user entered. It is important that the ranges are broken up with non-overlapping limiters or else the program will continue through, which will be explained more in-depth at the end. Once that range is located or if the subject exists on only one shelf the program loads the corresponding graphic in a new tab and updates the flavor text to say [call number] is on shelf [shelf number]. You will need to create and upload your own graphics, a template is provided in the main directory of the GitHub. The file type is a GIMP save format.

**Things To Keep In Mind When Editing The Program For Use At Your Institution**

* This is setup for the Library of Congress Classification System (though as long as you have a system that has a letter preceding a number this tool should still work for you). If you use a different system you may need to change some of the if statements to more closely align to your needs.
* Any reference to the index of the list of letters (or the spot in the list) should be -1 from your count since Python begins at 0 rather than 1. So, in our example earlier we used AX, which is 25th on our list but we remember its position as 24. This is also true when setting upper numeral values, such as when we set the max call number at 9,999 we had to tell python to start at 1 and stop at 10,000.
* Do not use else statements in the Python code. Be specific. This script acts fundamentally like a large while loop meaning until it is broken out of it will keep scanning, which means until the tab/pop-up is closed. If your step two code reads:

*if int(storage[‘letters’]) == 24:*

*if int(e.data) <= 42: “do A”*

*if in(e.data) <= 256: “do B”*

*else: “do C”*

and if the user said there call number was AX13 you would expect the program to just do A however since there is nothing stopping the code it will do A, then B, and then C since those are all valid by themselves (even though we read the else a computer will treat it as “since you made it this far those others must be false” even if that isn’t true. If your code regarding the letter makes that same mistake the program will open potentially hundreds of tabs. This is an unintentional side-effect from making it as “real time” as possible and a carryover from when the code went when the textbox was changed and not on a button press. It’s certainly possible to break this out into smaller functions to avoid this issue but rewriting the full length version of this tools was not high on the priority during the input conversion.

* The program is currently set up to just ignore invalid input patterns. If you prefer, it is possible to display invalid command warnings by adding “if !=” statements which only update the flavor, instruction, or hint text.
* By adding ‘or’ operators to if statements your code can be friendlier to end users (and yourself). This means inputs can be case insensitive or have multiple modes feed into a specific mode without that being visible. Just note that some code may require it may be easier to manually adjust letter cases.
* It is possible to make this search more specific by adding another mode/step or two which repeat callnum1 and callnum2 but for post “.” bits (e.g. callnum1/2: ***AF1956***.FN155vs. callnum3/4: AF1956.***FN155***). This base code does not go that specific as it would exceeds currently posted signage at AU. Moreover, in concept, it would be possible to make each book in a library notated down to it’s exact spot on a shelf by adding more layers of modes and if statements. It would also be possible to search by only using callnum1, as in looking for subjects in general (a mode that may be added in the future).
* Debug line numbers start at the beginning of each script, not the start of the document.
* Errors in the Python script can occur if you don’t properly indent and if you don’t maintain common indenting blocks (for example using spaces on one line and the tab key on the next)

Attached is a list of letters A-ZZ, formatted exactly as in the code, which can be used to quickly count position. Simply drag over the list and subtract 1 from the word count.

'A', 'AA', 'AB', 'AC', 'AD', 'AE', 'AF', 'AG', 'AH', 'AI', 'AJ', 'AK', 'AL', 'AM', 'AN', 'AO', 'AP', 'AQ', 'AR', 'AS', 'AT', 'AU', 'AV', 'AW', 'AX', 'AY', 'AZ', 'B', 'BA', 'BB', 'BC', 'BD', 'BE', 'BF', 'BG', 'BH', 'BI', 'BJ', 'BK', 'BL', 'BM', 'BN', 'BO', 'BP', 'BQ', 'BR', 'BS', 'BT', 'BU', 'BV', 'BW', 'BX', 'BY', 'BZ', 'C', 'CA', 'CB', 'CC', 'CD', 'CE', 'CF', 'CG', 'CH', 'CI', 'CJ', 'CK', 'CL', 'CM', 'CN', 'CO', 'CP', 'CQ', 'CR', 'CS', 'CT', 'CU', 'CV', 'CW', 'CX', 'CY', 'CZ', 'D', 'DA', 'DB', 'DC', 'DD', 'DE', 'DF', 'DG', 'DH', 'DI', 'DJ', 'DK', 'DL', 'DM', 'DN', 'DO', 'DP', 'DQ', 'DR', 'DS', 'DT', 'DU', 'DV', 'DW', 'DX', 'DY', 'DZ', 'E', 'EA', 'EB', 'EC', 'ED', 'EE', 'EF', 'EG', 'EH', 'EI', 'EJ', 'EK', 'EL', 'EM', 'EN', 'EO', 'EP', 'EQ', 'ER', 'ES', 'ET', 'EU', 'EV', 'EW', 'EX', 'EY', 'EZ', 'F', 'FA', 'FB', 'FC', 'FD', 'FE', 'FF', 'FG', 'FH', 'FI', 'FJ', 'FK', 'FL', 'FM', 'FN', 'FO', 'FP', 'FQ', 'FR', 'FS', 'FT', 'FU', 'FV', 'FW', 'FX', 'FY', 'FZ', 'G', 'GA', 'GB', 'GC', 'GD', 'GE', 'GF', 'GG', 'GH', 'GI', 'GJ', 'GK', 'GL', 'GM', 'GN', 'GO', 'GP', 'GQ', 'GR', 'GS', 'GT', 'GU', 'GV', 'GW', 'GX', 'GY', 'GZ', 'H', 'HA', 'HB', 'HC', 'HD', 'HE', 'HF', 'HG', 'HH', 'HI', 'HJ', 'HK', 'HL', 'HM', 'HN', 'HO', 'HP', 'HQ', 'HR', 'HS', 'HT', 'HU', 'HV', 'HW', 'HX', 'HY', 'HZ', 'I', 'IA', 'IB', 'IC', 'ID', 'IE', 'IF', 'IG', 'IH', 'II', 'IJ', 'IK', 'IL', 'IM', 'IN', 'IO', 'IP', 'IQ', 'IR', 'IS', 'IT', 'IU', 'IV', 'IW', 'IX', 'IY', 'IZ', 'J', 'JA', 'JB', 'JC', 'JD', 'JE', 'JF', 'JG', 'JH', 'JI', 'JJ', 'JK', 'JL', 'JM', 'JN', 'JO', 'JP', 'JQ', 'JR', 'JS', 'JT', 'JU', 'JV', 'JW', 'JX', 'JY', 'JZ', 'K', 'KA', 'KB', 'KC', 'KD', 'KE', 'KF', 'KG', 'KH', 'KI', 'KJ', 'KK', 'KL', 'KM', 'KN', 'KO', 'KP', 'KQ', 'KR', 'KS', 'KT', 'KU', 'KV', 'KW', 'KX', 'KY', 'KZ', 'L', 'LA', 'LB', 'LC', 'LD', 'LE', 'LF', 'LG', 'LH', 'LI', 'LJ', 'LK', 'LL', 'LM', 'LN', 'LO', 'LP', 'LQ', 'LR', 'LS', 'LT', 'LU', 'LV', 'LW', 'LX', 'LY', 'LZ', 'M', 'MA', 'MB', 'MC', 'MD', 'ME', 'MF', 'MG', 'MH', 'MI', 'MJ', 'MK', 'ML', 'MM', 'MN', 'MO', 'MP', 'MQ', 'MR', 'MS', 'MT', 'MU', 'MV', 'MW', 'MX', 'MY', 'MZ', 'N', 'NA', 'NB', 'NC', 'ND', 'NE', 'NF', 'NG', 'NH', 'NI', 'NJ', 'NK', 'NL', 'NM', 'NN', 'NO', 'NP', 'NQ', 'NR', 'NS', 'NT', 'NU', 'NV', 'NW', 'NX', 'NY', 'NZ', 'O', 'OA', 'OB', 'OC', 'OD', 'OE', 'OF', 'OG', 'OH', 'OI', 'OJ', 'OK', 'OL', 'OM', 'ON', 'OO', 'OP', 'OQ', 'OR', 'OS', 'OT', 'OU', 'OV', 'OW', 'OX', 'OY', 'OZ', 'P', 'PA', 'PB', 'PC', 'PD', 'PE', 'PF', 'PG', 'PH', 'PI', 'PJ', 'PK', 'PL', 'PM', 'PN', 'PO', 'PP', 'PQ', 'PR', 'PS', 'PT', 'PU', 'PV', 'PW', 'PX', 'PY', 'PZ', 'Q', 'QA', 'QB', 'QC', 'QD', 'QE', 'QF', 'QG', 'QH', 'QI', 'QJ', 'QK', 'QL', 'QM', 'QN', 'QO', 'QP', 'QQ', 'QR', 'QS', 'QT', 'QU', 'QV', 'QW', 'QX', 'QY', 'QZ', 'R', 'RA', 'RB', 'RC', 'RD', 'RE', 'RF', 'RG', 'RH', 'RI', 'RJ', 'RK', 'RL', 'RM', 'RN', 'RO', 'RP', 'RQ', 'RR', 'RS', 'RT', 'RU', 'RV', 'RW', 'RX', 'RY', 'RZ', 'S', 'SA', 'SB', 'SC', 'SD', 'SE', 'SF', 'SG', 'SH', 'SI', 'SJ', 'SK', 'SL', 'SM', 'SN', 'SO', 'SP', 'SQ', 'SR', 'SS', 'ST', 'SU', 'SV', 'SW', 'SX', 'SY', 'SZ', 'T', 'TA', 'TB', 'TC', 'TD', 'TE', 'TF', 'TG', 'TH', 'TI', 'TJ', 'TK', 'TL', 'TM', 'TN', 'TO', 'TP', 'TQ', 'TR', 'TS', 'TT', 'TU', 'TV', 'TW', 'TX', 'TY', 'TZ', 'U', 'UA', 'UB', 'UC', 'UD', 'UE', 'UF', 'UG', 'UH', 'UI', 'UJ', 'UK', 'UL', 'UM', 'UN', 'UO', 'UP', 'UQ', 'UR', 'US', 'UT', 'UU', 'UV', 'UW', 'UX', 'UY', 'UZ', 'V', 'VA', 'VB', 'VC', 'VD', 'VE', 'VF', 'VG', 'VH', 'VI', 'VJ', 'VK', 'VL', 'VM', 'VN', 'VO', 'VP', 'VQ', 'VR', 'VS', 'VT', 'VU', 'VV', 'VW', 'VX', 'VY', 'VZ', 'W', 'WA', 'WB', 'WC', 'WD', 'WE', 'WF', 'WG', 'WH', 'WI', 'WJ', 'WK', 'WL', 'WM', 'WN', 'WO', 'WP', 'WQ', 'WR', 'WS', 'WT', 'WU', 'WV', 'WW', 'WX', 'WY', 'WZ', 'X', 'XA', 'XB', 'XC', 'XD', 'XE', 'XF', 'XG', 'XH', 'XI', 'XJ', 'XK', 'XL', 'XM', 'XN', 'XO', 'XP', 'XQ', 'XR', 'XS', 'XT', 'XU', 'XV', 'XW', 'XX', 'XY', 'XZ', 'Y', 'YA', 'YB', 'YC', 'YD', 'YE', 'YF', 'YG', 'YH', 'YI', 'YJ', 'YK', 'YL', 'YM', 'YN', 'YO', 'YP', 'YQ', 'YR', 'YS', 'YT', 'YU', 'YV', 'YW', 'YX', 'YY', 'YZ', 'Z', 'ZA', 'ZB', 'ZC', 'ZD', 'ZE', 'ZF', 'ZG', 'ZH', 'ZI', 'ZJ', 'ZK', 'ZL', 'ZM', 'ZN', 'ZO', 'ZP', 'ZQ', 'ZR', 'ZS', 'ZT', 'ZU', 'ZV', 'ZW', 'ZX', 'ZY', 'ZZ'