autoHSP - script environment

WARNING: You should ONLY use this script as a script to run. This script is not fit for running in the notebook environment. There will be abnormalities.
NOTE: When creating a new template/run script with this template, you should uncheck the `Pause on Warnings` options and change the timeout constraint to 5 hours.

```
If[True,
 Pause[1];
 If[
  $NotebookPage[TimeConstraint] < 4 Hour || Not[$NotebookPage[IgnoreWarnings]],
  PauseScript[];
  Return["You must increase the `TimeConstraint`
     to 5 hours and turn off `Pause on Warnings`."]
 ]
]
```

Utility functions for logging

```
initVarGlobalLogMgr[reinit_ : False] := Block[{},
   If [! ValueQ[globalLogMgr] || reinit,
     globalLogMgr = {
        {{"Output", "perm"}, {"Welcome to the HSP script developed by sijie."}}
        (* tag, content *)
       };
    ];
  ];

initVarGlobalLogMgr[False];

getLogWithStyle[include_ : Nothing,
   exclude_ : Nothing, slice_ : All, returnPositionList_ : False,
   includeHow_ : SubsetQ, excludeHow_ : DisjointQ] := Block[
   {shouldHave, shouldNotHave, labels, logs, positions},
   shouldHave = If[ListQ[include], include, {include}];
   shouldNotHave = If[ListQ[exclude], exclude, {exclude}];
   positions = Range[Length[globalLogMgr]];
   labels = Quiet@Check[If[ListQ[#], #, {#}] & /@ globalLogMgr[[All, 1]], {}];
   positions =
    Quiet@Check[Select[positions, includeHow[labels[[#]], shouldHave] &], {}];
   positions =
    Quiet@Check[Select[positions, excludeHow[labels[[#]], shouldNotHave] &], {}];
```

```
    positions = Quiet@Check[positions[[slice]], {}];
    logs = Quiet@Check[globalLogMgr[[positions]][[All, 2]], {}];
    logs = Quiet@Check[If[ListQ[#], StringJoin[ToString /@ #], If[StringQ[#],
           #, ToString[#]]], " --- Invalid input log --- "] & /@ logs;
    If[returnPositionList =!= True,
     Return[logs];,
     Return[positions];
    ];
   ];

printLogWithStyle[include_ : Nothing, exclude_ : Nothing, slice_ : All] := Block[{},
    Return[StringRiffle[getLogWithStyle[include, exclude, slice, False], "\n"]];
   ];

delLogWithStyle[include_ : Nothing, exclude_ : Nothing, slice_ : All] :=
  Block[{positions},
   positions = getLogWithStyle[include, exclude, slice, True];
   globalLogMgr = Quiet@Delete[globalLogMgr, List /@ positions];
  ];

InNotebookEnvQ[check_ : Cells] := Block[{},
   Return[Quiet@Check[ListQ[check[]], False]];
  ];

InScriptEnvQ[check_ : Cells] := Block[{},
   Return[NotebookPage[Type] === Object[Notebook, Script] &&
       Not[InNotebookEnvQ[check]]];
  ];

(* use the `smartCellPrint` function
 to label your prints for future removals *)
smartCellPrint[objs_, labels_ : "tmp", styles_ : "Output", updateLog_ : True] :=
  Block[{content, cellTypes},
   content = StringJoin[ToString /@ If[ListQ[objs], objs, {objs}]];
   cellTypes = Join[ToString[#] & /@ If[ListQ[styles], styles, {styles}],
     ToString[#] & /@ If[ListQ[labels], labels, {labels}]];
   If[updateLog === True, AppendTo[globalLogMgr, {cellTypes, content}];];
   If[InNotebookEnvQ[], (* in a notebook environment *)
    CellPrint@Cell[BoxData[content], Sequence @@ cellTypes];,
    Return[content];
   ];
  ];

getCellWithStyle[include_ : Nothing, exclude_ : Nothing, slice_ : All] :=
```

```
  Block[{shouldHave, shouldNotHave, cells},
   shouldHave = If[ListQ[include], include, {include}];
   shouldNotHave = If[ListQ[exclude], exclude, {exclude}];
   cells = Cells[FrontEnd`EvaluationNotebook[]];
   cells = Quiet@
     Check[Select[cells, SubsetQ[CurrentValue[#, CellStyle], shouldHave] &], {}];
   cells = Quiet@Check[Select[cells,
       DisjointQ[CurrentValue[#, CellStyle], shouldNotHave] &], {}];
   Return[Quiet@Check[cells[[slice]], {}]];
  ];

delCellWithStyle[include_ : Nothing, exclude_ : Nothing, slice_ : All] := Block[{},
   If[InNotebookEnvQ[], (* in a notebook environment *)
    NotebookDelete[getCellWithStyle[include, exclude, slice]];
   ];
   delLogWithStyle[include, exclude, slice];
  ];

smartCellPrint[
  "Welcome to the HSP script developed by sijie.", "perm", "Output", False];
```

```
Welcome to the HSP script developed by sijie.
```

Functions to initialize variables

`globalSolvents`

`globalSolvents` stores all the solvents used for this experiment with a mapping

```
initVarGlobalSolvents[reinit_ : False] := Block[{},
   If[reinit === True,
    smartCellPrint[
      {"Warning: You are force-reloading `initVarGlobalSolvents`. This
          could cause unexpected behaviors. Make
          sure you know what you are doing."},
      Nothing, {"Message", "MSG", "reinit-warning"}];
   ];
   If[! ValueQ[globalSolvents] || reinit,
    globalSolvents = <|
       "S1" → Model[Sample, "Methyl Ethyl Ketone"],
       "S2" → Model[Sample, "Hexanes"],
       "S3" → Model[Sample, "Toluene, Reagent Grade"],
       "S4" → Model[Sample, "Propylene Carbonate"],
       "S5" → Model[Sample, "Acetonitrile, HPLC Grade"],
```

```
        "S6" → Model[Sample, "Methanol"],
        "S7" → Model[Sample, "Diacetone Alcohol, 98%"],
        "S8" → Model[Sample, "Cyclohexanol, 99%"],
        "S9" → Model[Sample, "Dimethylformamide, Reagent Grade"],
        "S10" → Model[Sample, "n-Butyl Acetate, ACS Grade"],
        "S11" → Model[Sample, "Ethanol, Reagent Grade"],
        "S12" → Model[Sample, "1-Propanol"],
        "S13" → Model[Sample, "Cyclohexane"],
        "S14" → Model[Sample,
          "Propylene Glycol 1-Monomethyl Ether 2-Acetate (PGMEA)"],
        "S15" → Model[Sample, "Tetrahydrofuran, Anhydrous"],
        "S16" → Model[Sample, "Dichloromethane, Reagent Grade"],
        "S17" → Model[Sample, "N-Methyl-2-pyrrolidone, Reagent Grade"],
        "S18" → Model[Sample, "gamma Butyrolactone"],
        "S19" → Model[Sample, "1,4-Dioxane, anhydrous"],
        "S20" → Model[Sample, "Ethylene Glycol Monomethyl Ether Acetate"],
        (* NOT in inventory *)
        "S21" → Model[Sample, "Diethylene Glycol Monoethyl Ether Acetate"],
        "S22" → Model[Sample, "Diethyl ether"],
        "S23" → Model[Sample, "Glycerol"],
        "S24" → Model[Sample, "Formamide"],
        "S25" → Model[Sample, "Milli-Q water"]
        |>;
    ];
    If[! ValueQ[globalSolventsUnits] || reinit,
     globalSolventsUnits = <|
        (* default unit for solvent is `Milliliter` *)
        "" → 1 Milliliter
        |>;
    ];
    If[! ValueQ[getSolventUnit[]] || reinit,
     getSolventUnit[code_ : ""] :=
       Lookup[globalSolventsUnits, code, globalSolventsUnits[""]];
    ];
   ];
```

`globalResins`

`globalResins` stores all the resins used for this experiment with a mapping

In[ ]:=

```
initVarGlobalResins[reinit_ : False] := Block[{},
   If[reinit === True,
    smartCellPrint[
      {"Warning: You are force-reloading `initVarGlobalResins`. This
```

```
        could cause unexpected behaviors. Make
        sure you know what you are doing."},
     Nothing, {"Message", "MSG", "reinit-warning"}];
 ];
 If[! ValueQ[globalResins] || reinit,
  globalResins = <|
     "R1" → Model[Sample, "Desmodur N 3300"], (* NOT in inventory *)
     "R2" → Model[Sample, "Desmodur N 3500"], (* NOT in inventory *)
     "R3" → Model[Sample, "Desmodur N 31100"], (* NOT in inventory *)
     "R4" → Model[Sample, "Desmodur N 3400"], (* NOT in inventory *)
     "R5" → Model[Sample, "Desmodur N 100"], (* NOT in inventory *)
     "R6" → Model[Sample, "Desmodur N 3900"], (* NOT in inventory *)
     "R7" → Model[Sample, "Desmodur Z 2589"], (* solid, NOT in inventory *)
     "R8" → Model[Sample, "Desmophen NH 1220"],
     "R9" → Model[Sample, "Desmophen NH 1420"], (* NOT in inventory *)
     "R10" → Model[Sample, "Desmophen NH 1520"],
     "R11" → Model[Sample, "Desmophen 1100"],
     "R12" → Model[Sample, "Desmophen 1200"],
     "R-2" → Model[Sample, "Poly(Ethylene Glycol) - 8000 MW"],
     (* solid flakes, NOT in inventory,
     optional resin to use if none of the above resins are available *)
     "R-1" → Model[Sample, "Polysorbate 20"](* viscous fluid (250-450 mPa.s),
     default resin to use if none of the above resins are available *)
     |>;
 ];
 If[! ValueQ[globalResinsUnits] || reinit,
  globalResinsUnits = <|
     (* By default, resins also use `Milliliter` as the unit *)
     "" → 1 Milliliter,
     "R7" → 1 Gram,
     (* Note that you can also change to `1.2Gram` for scaling *)
     "R-2" → 1 Gram
     |>;
 ];
 If[! ValueQ[getResinUnit[]] || reinit,
  getResinUnit[code_ : ""] :=
    Lookup[globalResinsUnits, code, globalResinsUnits[""]];
 ];
];
```

`globalVessel`

`globalVessel` stores information about the container that will be used. The following criteria should be met:

- A total of ~10 mL of liquid will be added to the vessel.
- The vessel should hold at least 15 mL of liquid. Preferably larger than 20 mL.
- The vessel should be compatible with the following shaker/mixer of choice.
- Clear glass walls (no texts, no marks, etc) and pure color caps (no horizontal textures).

```
initVarGlobalVessel[reinit_ : False, checkShaker_ : True] := Block[{},
   If[reinit === True,
    smartCellPrint[
      {"Warning: You are force-reloading `initVarGlobalVessel`. This
         could cause unexpected behaviors. Make
         sure you know what you are doing."},
      Nothing, {"Message", "MSG", "reinit-warning"}];
   ];

   If[! ValueQ[globalVessel] || reinit,
    (* if any of the variables not already defined or force `reinit`*)
    globalVessel = Model[Container, Vessel,
      "20mL Glass Scintillation Vial, With Cone-Shaped Cap Liner"];
    If[reinit && checkShaker && ValueQ[initVarGlobalShaker[]] &&
      ValueQ[globalShaker], initVarGlobalShaker[reinit, globalVessel];];
   ];
  ];
```

`globalShaker`, `globalShakeTime`, etc.

You need to make sure that the shaker chosen here is compatible with the previous `globalVessel`.
A quick way to check this is by using `AnyTrue[MixDevices[globalVessel, 10 Milliliter], # == globalShaker &]`.
If you need to suppress the vessel-shaker check, pass `False` to `checkVessel`. Otherwise by default, `checkVessel` will try to find `globalVessel` and use it.

```
initVarGlobalShaker[reinit_ : False, checkVessel_ : True] := Block[{vessel},
   If[reinit === True,
    smartCellPrint[
      {"Warning: You are force-reloading `initVarGlobalShaker`. This
         could cause unexpected behaviors. Make
         sure you know what you are doing."},
      Nothing, {"Message", "MSG", "reinit-warning"}];
   ];

   If[! ValueQ[globalShaker] || reinit,
    (* if any of the variables not already defined or force `reinit`*)
    (* globalShaker=Model[Instrument,Vortex,"20 mm Bottle Vortex Genie"]; *)
    (* globalShaker=
      Model[Instrument,Shaker,"Burrell Scientific Wrist Action Shaker"]; *)
```

```
 globalShaker = Model[Instrument, Roller, "Enviro-Genie"];

 (* Shaker-vessel compatibility check *)
 vessel = False;
 (* we will not check the shaker-
  vessel compatibility unless checkVessel is properly configured *)
 If[checkVessel === True,
  If[AllTrue[{globalVessel}, ValueQ],
    vessel = globalVessel;,
    smartCellPrint[
      "`checkVessel` is set to True but `globalVessel` has not been set.",
      "initVarGlobalShaker"];
   ];
 ];
 If[Quiet@Check[checkVessel[Type] === Model[Container, Vessel], False],
  vessel = checkVessel;,
  If[checkVessel =!= False && checkVessel =!= True,
    smartCellPrint["Wrong argument passed to `checkVessel`. Shoulde be
        one of True|False|Model[Container,Vessel,...]",
      "initVarGlobalShaker"];
   ];
 ];

 If[Not[vessel === False],
  smartCellPrint[
   "Checking for shaker-vessel compatibility... This might take
     some time...", "initVarGlobalShaker-incompatible"];
  If[AnyTrue[
    MixDevices[vessel, vessel[MaxVolume] / 2], # === globalShaker[Object] &],
   smartCellPrint[
     {"Success: shaker `", globalShaker, "` is compatible with vessel `",
      vessel, "`."}, "initVarGlobalShaker"];,
   smartCellPrint[{"Warning: the shaker `",
      globalShaker, "` is not compatible with your vessel `", vessel,
      "`. Please match these choices properly."}, "initVarGlobalShaker"];
  ];,
  smartCellPrint["Shaker-vessel compatibility test NOT performed.",
    "initVarGlobalShaker"];
 ];

 delCellWithStyle["initVarGlobalShaker-incompatible"];
]; (* shaker initiation complete *)

If[! ValueQ[globalShakeTime] ||
```

```
     ! ValueQ[globalShakeTimeSolvent] || ! ValueQ[globalShakeRate] || reinit,
    globalShakeTime = 12 Hour; (* How long to mix the sample *)
    globalShakeTimeSolvent = 2 Hour;
    globalShakeRate = 15 RPM;(* This might not be applicable to the shaker*)
   ];
   If[! ValueQ[globalSitTime] || reinit,
    globalSitTime = 24 Hour;
    (* how long to sit the sample for before taking the image *)
   ];
  ];
```

`globalCamera`

In[ ]:=
```
initVarGlobalCamera[reinit_ : False] := Block[{},
   If[reinit === True,
    smartCellPrint[
      {"Warning: You are force-reloading `initVarGlobalCamera`. This
          could cause unexpected behaviors. Make
          sure you know what you are doing."},
      Nothing, {"Message", "MSG", "reinit-warning"}];
   ];

   If[! ValueQ[globalCamera] || ! ValueQ[globalCameraShowVessel] || reinit,
    (* if any of the variables not already defined or force `reinit`*)
    globalCamera =
     Model[Instrument, SampleImager, "Emerald DSLR Camera Imaging Station"];
    globalCameraShowVessel = True(* when imaging, image container or not *)
   ];
  ];
```

`globalTaskMgr` and `globalLogMgr`

`globalTaskMgr` will serve as a collection of all tasks (“prep”, “image”, “EOE”) claimed by different threads running. For example:

```
globalTaskMgr = <|
 “thread1” -> <|
  “task” -> “prep”,(* prepare samples *)
  “taskId” -> “xxxxxx”,
  “samples” -> {
   <|”resin” -> “R1”, “solvent” -> {“S1”}, “ramount” -> 5, “samount” -> {5}, “label” -> “R1S1”|>,
   <|”resin” -> “R1”, “solvent” -> {“S7”}, “ramount” -> 5, “samount” -> {5}, “label” -> “R1S7”|>
   }
  |>,
 “thread2” -> <|
  “task” -> “image”,(* image samples *)
```

```
  “taskId” -> “xxxxxx”,
  “samples” -> {
    “R1S1”, “R1S7”
    }
  |>
|>
```

In[ ]:=
```
initVarGlobalTaskMgr[reinit_ : False, forsure_ : False] := Block[{},
   If[reinit === True,
    smartCellPrint[
      {"Warning: You are force-reloading `initVarGlobalTaskMgr`. This
         could cause unexpected behaviors. Make
         sure you know what you are doing."},
      Nothing, {"Message", "MSG", "reinit-warning"}];
   ];

   If[! ValueQ[globalTaskMgr] || And[reinit, forsure],
    (* if any of the variables not already defined or force `reinit`*)
    (* You really should NOT reinit `globalTaskMgr` after the script starts *)
    globalTaskMgr = <|
        (* `thread` → `task` *)
        |>;
   ];

   If[! ValueQ[getTaskForThread[]] || ! ValueQ[delTaskForThread[]] || reinit,
    getTaskForThread[thread_] := (
      If[! ValueQ[globalTaskMgr], globalTaskMgr = <||>;];
      Return[Lookup[globalTaskMgr, thread, <||>]];
     );
    delTaskForThread[thread_] := (
      If[! ValueQ[globalTaskMgr], globalTaskMgr = <||>;];
      globalTaskMgr = KeyDrop[globalTaskMgr, thread];
      Return[globalTaskMgr];
     );
   ];

   If[! ValueQ[parseAssocList[]] || ! ValueQ[getPosOfLength[]] || reinit,
    (* if any of the variables not already defined or force `reinit`*)
    parseAssocList[key_, assocList_, fillMissWith_ : Nothing] := Cases[assocList,
      assoc_ /; AssociationQ[assoc] :> Lookup[assoc, key, fillMissWith]];
    getPosOfLength[listOfLists_, N_ : 1,
      filterFunc_ : GreaterEqual, lengthFunc_ : Length] := Select[
      Range[Length[listOfLists]], filterFunc[lengthFunc[listOfLists[[#]]], N] &]
   ];
  ];
```

`globalNextUrl`, `globalUploadUrl`, and `parseJsonResponse`

Variables for server communication
`parseJsonResponse`: Function to parse JSON response from HTTP request

In[ ]:=
```
initVarGlobalUrl[reinit_ : False] := Block[{},
   If[reinit === True,
    smartCellPrint[
      {"Warning: You are force-reloading `initVarGlobalUrl`. This could cause
         unexpected behaviors. Make sure you know what you are
         doing."}, Nothing, {"Message", "MSG", "reinit-warning"}];
   ];

   If[! ValueQ[globalDomainName] || ! ValueQ[globalApiKey] || reinit,
    (* if any of the variables not already defined or force `reinit`*)
    globalDomainName = "https://XXXXX.XXXXX.cmu.edu/api/";
    globalApiKey = "XXXXXXXXXXXXXXXXXXX"; (* API KEY *)
   ];

   If[! ValueQ[globalNextUrl] ||
     ! ValueQ[globalUploadUrl] || ! ValueQ[globalNotifyUrl] || reinit,
    globalNextUrl = URLBuild[{globalDomainName, globalApiKey, "next"}];
    globalUploadUrl = URLBuild[{globalDomainName, globalApiKey, "upload"}];
    globalNotifyUrl = URLBuild[{globalDomainName, globalApiKey, "notify"}];
   ];

   If[! ValueQ[globalMaxRuns] || ! ValueQ[globalWaitInterval] || reinit,
    globalMaxRuns = 10;
    (* try communicate with the server for a max of 10 times *)
    globalWaitInterval = 60;
    (* each failure will cause the program to pause before next try *)
   ];

   If[! ValueQ[parseJsonResponse[]] || reinit,
    parseJsonResponse[response_] := (
       ResourceFunction["ToAssociations"][
        ImportString[response["Body"], "JSON"]]
      );
   ];
  ];
```

Other global variables

In[ ]:=
```
initVarGlobalOthers[reinit_ : False] := Block[{},
   If[reinit === True,
    smartCellPrint[
      {"Warning: You are force-reloading `initVarGlobalOthers`. This
         could cause unexpected behaviors. Make
         sure you know what you are doing."},
      Nothing, {"Message", "MSG", "reinit-warning"}];
   ];

   If[! ValueQ[globalExpSite] || reinit,
    globalExpSite = Object[Container, Site, "ECL-CMU"];
   ];

   If[! ValueQ[globalSampleNamePrefix] || reinit,
    globalSampleNamePrefix = "Washburn_HSP_";
   ];

   If[
    ! ValueQ[addSampleNamePrefix[]] || ! ValueQ[delSampleNamePrefix[]] || reinit,
    addSampleNamePrefix[string_, prefix_ : True] := Block[{tmpPrefix},
      tmpPrefix = If[StringQ[prefix], prefix, If[ValueQ[globalSampleNamePrefix],
         ToString[globalSampleNamePrefix], Return[string];
         ""]];
      If[StringLength[tmpPrefix] == 0, Return[string]];
      Return[If[! StringStartsQ[string, tmpPrefix | "id:"],
        StringJoin[tmpPrefix, string], string]];
     ];
    delSampleNamePrefix[string_, prefix_ : True] := Block[{tmpPrefix},
      tmpPrefix = If[StringQ[prefix], prefix, If[ValueQ[globalSampleNamePrefix],
         ToString[globalSampleNamePrefix], Return[string];
         ""]];
      If[StringLength[tmpPrefix] == 0, Return[string]];
      Return[If[StringStartsQ[string, tmpPrefix],
        StringReplace[string, StartOfString ~~ tmpPrefix → ""], string]];
     ];
   ];
  ];
```

a wrapper to init all variables

In[•]:=
```
initAllVars[reinit_ : False] := (
   initVarGlobalSolvents[reinit];
   initVarGlobalResins[reinit];
   initVarGlobalVessel[reinit, False];
   (* will NOT check if the vessel if compatible with the shaker *)
   initVarGlobalShaker[reinit, globalVessel];
   (* will also check if the vessel if compatible with the shaker *)
   initVarGlobalCamera[reinit];
   initVarGlobalTaskMgr[reinit, False];
   (* you should really NOT reinit this variable *)
   initVarGlobalUrl[reinit];
   initVarGlobalOthers[reinit];
  );

(* initialize the variables *)
initAllVars[False];
(* delCellWithStyle[{"Message","MSG","reinit-warning"}]; *)
```

```
getLogWithStyle["initVarGlobalShaker"]
```

Functions for server communications

Function to pull the next experiments

Get the next task for a thread with defined abilities for the thread.

```
getNextTaskFromServer[thread_, ability_ : {"prep", "image"},
   extraOutputTag_ : Nothing, maxRuns_ : True, waitInterval_ : True] := Block[
   {tmpUrl, tmpUrlResponse, tmpCounter,
    tmpMaxRuns, tmpWaitInterval, tmpPrintCellStyle},
   initVarGlobalUrl[False];
   (* in case URL parameters are not initialized *)
   tmpMaxRuns = If[IntegerQ[maxRuns] && maxRuns ≥ 1, maxRuns, globalMaxRuns, 10];
   tmpWaitInterval = If[RealValuedNumberQ[waitInterval] && waitInterval > 0,
     waitInterval, globalWaitInterval, 60];
   tmpPrintCellStyle = {"getNextTaskFromServer", If[StringQ[extraOutputTag] &&
       StringLength[extraOutputTag] ≥ 1, extraOutputTag, Nothing, Nothing]};
   smartCellPrint[{"Trying to get the next task for thread `",
     thread, "` with abilities `", ability, "`" }, tmpPrintCellStyle];
   tmpUrl = URLBuild[globalNextUrl, <|"thread" → thread, "ability" → ability|>];
```

```
    Do[
     tmpUrlResponse = URLRead[tmpUrl];
     If[tmpUrlResponse["StatusCode"] == 200,
      (
       (* If GET request successful, update task manager *)
       smartCellPrint[{"Try ", tmpCounter, "/", tmpMaxRuns,
         ": Trying to get the next task for thread `", thread,
         "` with abilities `", ability, "`"}, tmpPrintCellStyle];
       globalTaskMgr[thread] = parseJsonResponse[tmpUrlResponse];
       Break[];
      ),
      (* if not successful, pause 30 minutes until next retry *)
      smartCellPrint[{"Try ", tmpCounter, "/",
        maxRuns, " failed with code ", tmpUrlResponse["StatusCode"],
        ". Waiting ", tmpWaitInterval, " seconds before the next try."},
       {"getNextTaskFromServer", globalSmartCellPrintTag}];
      If[tmpCounter < tmpMaxRuns, Pause[tmpWaitInterval];];
     ],
     {tmpCounter, Max[1, tmpMaxRuns]}
    ]; (* Do ends *)

    If[tmpUrlResponse["StatusCode"] ≠ 200,
     smartCellPrint[{tmpMaxRuns, " URL GET request(s) failed with code ",
       tmpUrlResponse["StatusCode"], ": ", tmpUrl}, tmpPrintCellStyle];
     Return[False];,
     delCellWithStyle[tmpPrintCellStyle];
     Return[True];
    ]
   ];
```

Function to upload an image

For example, `uploadSampleImage[“February 2024 User Training HPLC Sample”,”test”,3,1,False]`

```
uploadSampleImageToServer[sampleName_,
    taskId_, extraOutputTag_ : Nothing, validateTime_ : True,
    prefix_ : True, maxRuns_ : True, waitInterval_ : True] := Block[
    {prefixedSampleName, rawSampleName,
     tmpObjectSample, tmpObjectVessel, imageFromVessel, tmpSampleImage,
     tmpSampleImageTime, tmpSampleMixedN, tmpSampleMixedNTarget,
     tmpSampleMixedTime, tmpUrl, tmpPostData, tmpUrlResponse,
     tmpUrlResponseAsso, tmpMaxRuns, tmpWaitInterval, tmpPrintCellStyle},
    initVarGlobalUrl[False];
    (* in case URL parameters are not initialized*)
```

```
prefixedSampleName = addSampleNamePrefix[sampleName, prefix];
rawSampleName = delSampleNamePrefix[sampleName, prefix];
tmpMaxRuns = If[IntegerQ[maxRuns] && maxRuns ≥ 1, maxRuns, globalMaxRuns, 10];
tmpWaitInterval = If[RealValuedNumberQ[waitInterval] && waitInterval > 0,
  waitInterval, globalWaitInterval, 60];
tmpPrintCellStyle = {"uploadSampleImageToServer", If[StringQ[extraOutputTag] &&
    StringLength[extraOutputTag] ≥ 1, extraOutputTag, Nothing, Nothing]};

tmpObjectSample = Object[Sample, prefixedSampleName];
tmpObjectVessel = Object[Container, Vessel, prefixedSampleName];
(* issue in binding appereance log? *)
If[Length[tmpObjectVessel[AppearanceLog]] == 0 &&
  Length[tmpObjectSample[AppearanceLog]] == 0,
 smartCellPrint[{"Sample `", prefixedSampleName, "` from task `",
   taskId, "` does NOT yet have image data."}, tmpPrintCellStyle];
 Return[False];(* appearance data not ready yet *)
];
If[Length[tmpObjectVessel[AppearanceLog]] > 0 &&
  Length[tmpObjectSample[AppearanceLog]] > 0,
 imageFromVessel = If[tmpObjectVessel[AppearanceLog][[-1]][[1]] ≥
     tmpObjectSample[AppearanceLog][[-1]][[1]], True, False, False];,
 imageFromVessel =
   If[Length[tmpObjectSample[AppearanceLog]] == 0, True, False, True];
];

tmpSampleImage = If[imageFromVessel, tmpObjectVessel[AppearanceLog][[-1]][[2]][
   Image], tmpObjectSample[AppearanceLog][[-1]][[2]][Image]];
If[Head[tmpSampleImage] == Image,
 tmpSampleImageTime = If[imageFromVessel, tmpObjectVessel[AppearanceLog][[-1]][[
    1]], tmpObjectSample[AppearanceLog][[-1]][[1]]],
 (* this should be the timestamp for the appearance *)
 smartCellPrint[{"Sample `", prefixedSampleName, "` from task `",
   taskId, "` does NOT yet have image data."}, tmpPrintCellStyle];
 Return[False];(* appearance data not ready yet *)
];

(* Image was successfully recorded,
next check if the image was taken after mixing *)
tmpSampleMixedN = 0;
tmpSampleMixedTime = tmpObjectSample[SampleHistory][[-1]][Date];

Do[
 If[Head[tmpOperation] == Mixed ||
   Quiet@Check[tmpOperation[Type] === Object[Protocol, Incubate], False],
```

```
  tmpSampleMixedN += 1;
  tmpSampleMixedTime = tmpOperation[DateCompleted];
 ], {tmpOperation, tmpObjectSample[Protocols]}
];
(* check if the sample was actually mixed *)
tmpSampleMixedNTarget = 1 +
  Boole[Length[StringCases[rawSampleName, RegularExpression["S\\d+"]]] > 1];
If[tmpSampleMixedN < tmpSampleMixedNTarget && validateTime === True,
 smartCellPrint[{"Sample `", prefixedSampleName,
   "` from task `", taskId, "` has not been mixed properly for ",
   tmpSampleMixedNTarget, " times."}, tmpPrintCellStyle];
 Return[False];
];

(* check if the image time was after the mixed time *)
If[tmpSampleImageTime - tmpSampleMixedTime < globalSitTime &&
  validateTime === True,
 smartCellPrint[{"Sample `", prefixedSampleName, "` from task `",
   taskId, "` has not yet been imaged after a sit time of ",
   globalSitTime, " after mixing."}, tmpPrintCellStyle];
 Return[False];
];

(* All things checked, this is a valid sample image. Now upload it. *)
initVarGlobalUrl[False]; (* just in case *)
tmpUrl = URLBuild[globalUploadUrl, <|"sample" → rawSampleName,
   "sid" → tmpObjectSample[ID], "tid" → taskId|>];
tmpPostData = <|
  Method → "POST",
  "Body" → {
    "file" → <|
      "Content" → ExportByteArray[tmpSampleImage, "JPG"],
      "MIMEType" → "image/jpg",
      "Name" → ToString[rawSampleName] <> ".jpg"
      |>
   }
  |>;

smartCellPrint[{"Image successfully retrieved ",
  If[validateTime === True, "and ", "but NOT "],
  "checked if its timestamp is valid. Trying to upload sample image of `",
  prefixedSampleName, "` under task id `", taskId,
  "` to the server."}, tmpPrintCellStyle];
Do[
```

```
  smartCellPrint[{"Try ", tmpCounter, "/", tmpMaxRuns,
    ": Trying to upload sample image of `", prefixedSampleName,
    "` under task id `", taskId, "` to the server."}, tmpPrintCellStyle];
  tmpUrlResponse = URLRead[HTTPRequest[tmpUrl, tmpPostData]];
  If[tmpUrlResponse["StatusCode"] == 200,
   tmpUrlResponseAsso = parseJsonResponse[tmpUrlResponse];
   If[AssociationQ[tmpUrlResponseAsso] && Length[tmpUrlResponseAsso] ≥ 1 &&
     StringEndsQ[Keys[tmpUrlResponseAsso][[1]], ".jpg"],
    Break[];
   ];
  ];
  (* if not successful, pause 30 minutes until next retry *)
  smartCellPrint[{"Try ", tmpCounter, "/", tmpMaxRuns, " failed with code ",
    tmpUrlResponse["StatusCode"], ". Waiting ", tmpWaitInterval,
    " seconds before the next try."}, tmpPrintCellStyle];
  If[tmpCounter < tmpMaxRuns, Pause[tmpWaitInterval];];,
  {tmpCounter, tmpMaxRuns}
 ];
 If[tmpUrlResponse["StatusCode"] ≠ 200,
  smartCellPrint[{tmpMaxRuns, " URL POST request(s) failed with code ",
    tmpUrlResponse["StatusCode"], ": ", tmpUrl}, tmpPrintCellStyle];
  Return[False];,
  delCellWithStyle[tmpPrintCellStyle];
  Return[True];
 ]
];
```

Function to notify the server about experimentation status

```
notifyServer[params_, extraOutputTag_ : Nothing,
   maxRuns_Integer : True, waitInterval_ : True] := Block[
   {tmpUrl, tmpUrlResponse, tmpMaxRuns, tmpWaitInterval, tmpPrintCellStyle},
   initVarGlobalUrl[False]; (* just in case *)
   tmpUrl = If[AssociationQ[params], URLBuild[globalNotifyUrl, params], params];
   tmpMaxRuns = If[IntegerQ[maxRuns] && maxRuns ≥ 1, maxRuns, globalMaxRuns, 10];
   tmpWaitInterval = If[RealValuedNumberQ[waitInterval] && waitInterval > 0,
     waitInterval, globalWaitInterval, 60];
   tmpPrintCellStyle = {"notifyServer", If[StringQ[extraOutputTag] &&
       StringLength[extraOutputTag] ≥ 1, extraOutputTag, Nothing, Nothing]};

   Do[
    smartCellPrint[{"Try ", tmpCounter, "/", tmpMaxRuns,
      ": Trying to reach the server at: ", tmpUrl}, tmpPrintCellStyle];
    tmpUrlResponse = URLRead[tmpUrl];
    If[tmpUrlResponse["StatusCode"] == 200,
     Break[];,
     (* if not successful, pause 30 minutes until next retry *)
     smartCellPrint[{"Try ", tmpCounter, "/", tmpMaxRuns, " failed with code ",
       tmpUrlResponse["StatusCode"], ". Waiting ", tmpWaitInterval,
       " seconds before the next try."}, tmpPrintCellStyle];
     If[tmpCounter < tmpMaxRuns, Pause[tmpWaitInterval];];
    ];,
    {tmpCounter, tmpMaxRuns}
   ];

   delCellWithStyle[tmpPrintCellStyle];
   If[tmpUrlResponse["StatusCode"] ≠ 200,
    smartCellPrint[{tmpMaxRuns, " runs to reach the server failed with code ",
      tmpUrlResponse["StatusCode"], ": ", tmpUrl}, tmpPrintCellStyle];
    Return[False];,
    Return[True];
   ];
  ];
```

Functions to run experiments

Function to prepare sample solutions

The input to this function should be the Association coming from globalTaskMgr[“thread***”]

```
prepSamples[tasks_, extraOutputTag_ : Nothing, prefix_ : True] := Block[
   {falseReturnVal, intendedTask, resins,
```

```
 resinsAmounts, solvents, solventsAmounts, labels, taskList,
 prepProtocol, solventsIndices, validExpFlag, tmpPrintCellStyle},
(* You should have initiated some of
 the global variables before you call this function *)
falseReturnVal = {False, False, False};
(* return this if an error occurs, {`status`,`protocol`,`samples`} *)
tmpPrintCellStyle = {"prepSamples", If[StringQ[extraOutputTag] &&
    StringLength[extraOutputTag] ≥ 1, extraOutputTag, Nothing, Nothing]};
smartCellPrint[
 {"Initiating function to build a prep sample protocol for tasks:", tasks},
 tmpPrintCellStyle];
intendedTask = Lookup[tasks, "task", ""];
If[! StringMatchQ[intendedTask, "prep", IgnoreCase → True],
 smartCellPrint[
  {"Wrong `prepSamples` function called to excute your intended task `",
   intendedTask, "`."}, tmpPrintCellStyle];
 Return[falseReturnVal];
];

(* preprocess requests in tasks *)
resins = parseAssocList["resin", tasks["samples"], False];
resinsAmounts = parseAssocList["ramount", tasks["samples"], False];
solvents = parseAssocList["solvent", tasks["samples"], False];
solventsAmounts = parseAssocList["samount", tasks["samples"], False];
labels = parseAssocList["label", tasks["samples"], False];
taskList = {resins, resinsAmounts, solvents, solventsAmounts, labels};
If[AnyTrue[taskList, MemberQ[#, False] &],
 smartCellPrint[
  {"Error: Your input tasks have missing entries in its members."},
  tmpPrintCellStyle];
 Return[falseReturnVal];
];

labels = addSampleNamePrefix[#, prefix] & /@ labels;
(* These labels will be `UploadName` names if not ids *)
If[! DuplicateFreeQ[labels],
 smartCellPrint[{"Error: Duplicated labels detected."}, tmpPrintCellStyle];
 Return[falseReturnVal];
];

If[Not[AllTrue[Flatten[resins], KeyExistsQ[globalResins, #] &]] ||
  Not[AllTrue[Flatten[solvents], KeyExistsQ[globalSolvents, #] &]],
 smartCellPrint[
  {"Error: Undefined resin(s) or solvent(s) found. Please check
```

```
      your inputs or update you `globalResins`
      and/or `globalSolvents`."}, tmpPrintCellStyle];
 Return[falseReturnVal];
];

(* Start manual sample preparation protocol *)
smartCellPrint[{"A valid set of tasks"}, tmpPrintCellStyle];
Quiet@Check[
  prepProtocol = ManualSamplePreparation[
      (* 1. Label containers *)
      LabelContainer[
       Label → labels,
       Container → Table[globalVessel, Length[labels]]
      ],

      (* 2. Sequentially add the solvents into the vessels; note that some
       only have 1 solvent, while the others might have 2, 3, ... *)
      Sequence @@ Reap[
         Do[
          solventsIndices =
           getPosOfLength[solvents, nSolvents, GreaterEqual, Length];
          If[
           Length[solventsIndices] < 1,
           Sow[Nothing]; Break[];(* Upper bound reached *)
          ];
          Sow[
           Transfer[
            Source → Lookup[globalSolvents,
              solvents⟦solventsIndices⟧⟦All, nSolvents⟧],
            Destination → labels⟦solventsIndices⟧,
            Amount → solventsAmounts⟦solventsIndices⟧⟦All, nSolvents⟧ *
              getSolventUnit[solvents⟦solventsIndices⟧⟦All, nSolvents⟧]
           ]
          ],
          {nSolvents, 10}
          (* using 10 solvents would be crazy if it ever happens*)
         ]
        ][[2, 1]],

      (* 2+. Do we need to mix
        the samples with more than 1 solvent in them? *)
      solventsIndices = getPosOfLength[solvents, 2, GreaterEqual, Length];
      If[Length[solventsIndices] > 0,
       Mix[
```

```
        Sample → labels[[solventsIndices]],
        Instrument → globalShaker,
        MixUntilDissolved → False, (* Even though
         we'd want the solvents to dissolve with each other *)
        Time → globalShakeTimeSolvent,
        MixRate → globalShakeRate,
        ImageSample → True,
        MeasureVolume → False,
        MeasureWeight → False
       ],
       Nothing (* Do nothing if all samples require at most 1 solvent *)
      ],

      (* 3. Transfer resin after solvent *)
      Transfer[
       Source → Lookup[globalResins, resins],
       Destination → labels,
       Amount → resinsAmounts * getResinUnit[resins]
      ],

      (* 4. Shake the resin-solvent mixture *)
      Mix[
       Sample → labels,
       Instrument → globalShaker,
       MixUntilDissolved → False, (* Dissolving is not guaranteed *)
       Time → globalShakeTime,
       MixRate → globalShakeRate
      ],

      ImageSample → True,
      MeasureVolume → False,
      MeasureWeight → False
     ];,
   smartCellPrint[
    {"Some unexpected error occurred when building the prep sample
       protocol. Please check again."}, tmpPrintCellStyle];
   Return[falseReturnVal];
  ];

delCellWithStyle[tmpPrintCellStyle];
smartCellPrint[{"Starting to validate the generated prep sample protocol: ",
  prepProtocol}, tmpPrintCellStyle];
validExpFlag = Check[
  prepProtocol = ExperimentSamplePreparation[
```

```
        prepProtocol,
        ImageSample → True,
        MeasureVolume → False,
        MeasureWeight → False,
        OptimizeUnitOperations → True,
        Site → globalExpSite
       ];
      True,
      False
     ];
    If[validExpFlag =!= True,
     smartCellPrint[
       {"Failed to generate a valid prep sample protocol for tasks: ", tasks},
       tmpPrintCellStyle];,
     delCellWithStyle[tmpPrintCellStyle, Nothing, -1];
    ];
    Return[{validExpFlag, prepProtocol, labels}];
   ];
```

Function to upload sample names to the constellation from a prep sample protocol

```
rememberSamples[protocol_, samples_,
    taskId_ : "", extraOutputTag_ : Nothing, prefix_ : True] := Block[
    {whichProtocol, returnVal, names, samplesObjects, i, tmpPrintCellStyle},
    tmpPrintCellStyle = {"rememberSamples", If[StringQ[extraOutputTag] &&
        StringLength[extraOutputTag] ≥ 1, extraOutputTag, Nothing, Nothing]};
    whichProtocol = If[StringQ[protocol],
      Object[Protocol, ManualSamplePreparation, protocol], protocol];
    If[Head[taskId] === String && StringLength[taskId] > 0,
     Quiet@UploadName[whichProtocol, addSampleNamePrefix[taskId]];
    ];

    names = If[ListQ[samples], samples, {samples}];
    names = addSampleNamePrefix[#, prefix] & /@ names;
    (* These labels should be `UploadName` names if not ids *)

    smartCellPrint[{"Got inputs protocol `", protocol, "` and samples `", names,
      "`. Now trying to upload the sample names..."}, tmpPrintCellStyle];
    samplesObjects = Quiet@LookupLabeledObject[whichProtocol, names];
    returnVal = # === Null & /@ samplesObjects;
    For[i = 1, i ≤ Length[names], i++,
     returnVal[[i]] = Check[ListQ[UploadName[samplesObjects[[i]], names[[i]]]], False];
    ];
```

```
  (* Print some error messages *)
  For[i = 1, i ≤ Length[names], i++,
   If[returnVal[[i]] == False,
     If[samplesObjects[[i]] === Null,
       smartCellPrint[{"Error: Could not find name `", names[[i]],
         "` in protocol `", protocol, "`."}, tmpPrintCellStyle];
       smartCellPrint[{"Error: Could not upload name `", names[[i]],
         "` for the found sample object `", samplesObjects[[i]],
         "` in protocol `", protocol, "`."}, tmpPrintCellStyle];
      ];,
     (* the sample name was uploaded successfully →
      Now upload its sample content and source protocol *)
     If[samplesObjects[[i]][Type] === Object[Sample],
      UploadName[samplesObjects[[i]][Container], names[[i]]];
     ];
     If[samplesObjects[[i]][Type] === Object[Container, Vessel],
      UploadName[samplesObjects[[i]][ContentsLog][[-1]][[3]], names[[i]]];
     ];
    ];
  ];

  If[AllTrue[returnVal, TrueQ], delCellWithStyle[tmpPrintCellStyle];];
  Return[returnVal];
 ];
```

Function to image samples

```
imageSamples[tasks_, extraOutputTag_ : Nothing, prefix_ : True] := Block[
   {falseReturnVal, intendedTask,
    samples, imageProtocol, validExpFlag, tmpPrintCellStyle},
   falseReturnVal = {False, False, False};
   (* return this if an error occurs, {`status`,`protocol`,`samples`} *)
   tmpPrintCellStyle = {"imageSamples", If[StringQ[extraOutputTag] &&
       StringLength[extraOutputTag] ≥ 1, extraOutputTag, Nothing, Nothing]};
   smartCellPrint[
    {"Initiating function to build a image sample protocol for tasks:", tasks},
    "imageSamples"];

   intendedTask = Lookup[tasks, "task", ""];
   If[! StringMatchQ[intendedTask, "image", IgnoreCase → True],
    smartCellPrint[
     {"Wrong `imageSamples` function called to excute your intended task `",
```

```
    intendedTask, "`."}, tmpPrintCellStyle];
 Return[falseReturnVal];
];
samples = Lookup[tasks, "samples", {}];
samples = If[! ListQ[samples], {samples}, samples];
samples = addSampleNamePrefix[#, prefix] & /@ samples;
(* These labels should be `UploadName` names if not ids *)
If[Not[DuplicateFreeQ[samples]],
 smartCellPrint[{"Found dupicates in your parsed sample names: `",
   samples, "`. Please check again."}, tmpPrintCellStyle];
 Return[falseReturnVal];
];

(* Next, search for these samples and image them *)
smartCellPrint[{"Starting to validate the generated image sample protocol: ",
  imageProtocol}, tmpPrintCellStyle];
validExpFlag = Check[
  ValidExperimentImageSampleQ[
    Object[Sample, #] & /@ samples,
    Instrument → globalCamera,
    ImageContainer → globalCameraShowVessel,
    Site → globalExpSite
   ] === True,
  smartCellPrint[
   {"Some unexpected error occurred when building the image sample
      protocol. Probabaly because one or more of the
      sample names are not correctly recognized. Please
      check again.", samples}, tmpPrintCellStyle];
  Return[falseReturnVal];
  False
 ];

If[validExpFlag =!= True,
 smartCellPrint[
  {"Failed to generate a valid image sample protocol for tasks: ", tasks},
  tmpPrintCellStyle];
 Return[falseReturnVal];
];

delCellWithStyle[tmpPrintCellStyle];
imageProtocol = ExperimentImageSample[
  Object[Container, Vessel, #] & /@ samples,
  Instrument → globalCamera,
  ImageContainer → globalCameraShowVessel,
```

```
    Site → globalExpSite
   ];

  Return[{validExpFlag, imageProtocol, samples}];
 ];
```

HSP - Main (putting things together)

Some preparatory work

```
thread = "thread1";
(* should not contain characters such as `-`, `@` and `:` *)
ability = {"prep", "image"};
validateImageTime = True;
extraOutputTag = "HSPMain:" <> thread;
prefix = globalSampleNamePrefix;(* Washburn_HSP_*)
maxRuns = globalMaxRuns;
waitInterval = globalWaitInterval;

whichCycle = 0;
useLastTask = False;
(* if last cycle's task went wrong and you need to rerun it *)
thisCycleWentFine = True;
(* if everything is executing without error/exception *)
EOE = False;(* end of experiment *)

(* pauseExecution will handle errors and exceptions *)
pauseExecution[
   params_ : <|"status" → "exception",
     "message" → "You attention is required ASAP."|>,
   extraOutputTag_ : Nothing, maxRuns_ : True, waitInterval_ : True] := (
   smartCellPrint[
    {"Pausing script execution due to error or exception.\nYou are
       advised to review the error information carefully and make
       adjustments on the server before resuming execution."},
    {"pauseExecution", extraOutputTag}];
   thisCycleWentFine = False;
   notifyServer[params, extraOutputTag, maxRuns, waitInterval];
   PauseScript[];
  );

printLineBreak[cycle_ : Nothing, extraOutputTag_ : Nothing] := (
   Return[smartCellPrint[{StringRepeat["-", 12], " ", cycle, " ",
       StringRepeat["-", 12]}, Flatten[{"perm", "line", extraOutputTag}]]];
  );

printLineBreak["START", extraOutputTag];
smartCellPrint[{"Start HSPMain (script) execution on thread `", thread, "`."},
  {"perm", extraOutputTag}];
```

Start cycle from here

In[ ]:=
```
Label["start:HSP"];
```

In[ ]:=
```
whichCycle += 1;
tasks = <||>;(* the set of tasks to be performed *)
taskType = "";(* which task will be performed in this cycle *)
extraOutputTag = "HSPMain:" <> thread <> "@" <> ToString[whichCycle];
thisCycleWentFine = True;(* a fresh new start,
you'd better behave well this time *)
thisTaskGotMatched = False;
failMessage = <|"status" → "exception",
   "message" → "An exception happened. You attention is required ASAP."|>;
printLineBreak[whichCycle, extraOutputTag];
```

1 - Check if there is already thread-claimed tasks in `globalTaskMgr`

this should rarely happen since before each `Goto`, the tasks are removed

```
If[thisCycleWentFine && And[KeyExistsQ[globalTaskMgr, thread], Not[useLastTask]],
  (* Why is this thread NOT reomved from task manager? *)
  smartCellPrint[{"You are trying to run thread `", thread,
    "` but it is already registered under `globalTaskMgr` with tasks: ",
    getTaskForThread[thread],
    ".\nDid you forget to `delTaskForThread`? You should also check if the task
      is already successfully performed in the lab.\nWARNING: make sure to
      update the server configuration before you skip the pause."},
   {"threadExistsException", extraOutputTag}];
  thisCycleWentFine = False;
  failMessage = <|"status" → "exception", "message" → "threadExistsException"|>;
 ];
```

2 - Get next set of tasks from the server

```
If[thisCycleWentFine,
 If[And[useLastTask, KeyExistsQ[globalTaskMgr, thread]],
  taskStatus = True,
  taskStatus = getNextTaskFromServer[thread,
     ability, extraOutputTag, maxRuns, waitInterval] === True
 ];
 If[Not[taskStatus],
  thisCycleWentFine = False;
  failMessage = <|"status" → "exception",
    "message" → "getNextTaskFromServerException"|>;
  Return[smartCellPrint[{"Cycle ", whichCycle,
     ": failed to get the next set of tasks from the server."},
    {"getNextTaskFromServer", extraOutputTag}]]
  ,
  tasks = getTaskForThread[thread];
  taskType = Lookup[tasks, "task", ""];
  If[! StringMatchQ[taskType, "pause", IgnoreCase → True],
   Return[smartCellPrint[
     {"Cycle ", whichCycle, ": got the next set of tasks from the server: ",
      tasks}, {"getNextTaskFromServer", extraOutputTag}]]
  ]
 ]
]
```

3.1 - prepSamples (?)

```
If[thisCycleWentFine && StringMatchQ[taskType, "prep", IgnoreCase → True],
 thisTaskGotMatched = True;
 urlParams = <|"status" → "exception", "message" → "unknownException"|>;

 {status, protocol, samples} = prepSamples[tasks, extraOutputTag];
 If[status =!= True,
  thisCycleWentFine = False;
  failMessage = <|"status" → "exception", "message" → "prepSamplesException"|>;
 ]
]
```

Supposedly, if the previous cell returns a valid protocol, it should automatically run. And the next cell will only start executing when that protocol has a status of completed.
NOTE: If the above cell (prepSamples) runs successfully, but the following cell (rememberSamples) failed. You should go to a notebook page and fix `rememberSamples` from there. DO NOT try to fix it in the script environment.

```
If[thisCycleWentFine && status && StringMatchQ[taskType,
    "prep", IgnoreCase → True] && protocol[Status] === Completed,
  status = rememberSamples[protocol,
    samples, Lookup[tasks, "taskId", ""], extraOutputTag];

  If[AllTrue[status, TrueQ],
   protocolProgress = protocol[CheckpointProgress];
   samplePrepEndTime = "";
   If[Quiet@ListQ[protocolProgress],
    Do[
      If[progress[[1]] === "Sample Preparation",
        samplePrepEndTime = DateString[progress[[3]], "ISODateTime"] <>
          DateString[progress[[3]], "ISOTimeZone"];
        Break[];
       ];
      , {progress, Reverse[protocolProgress]}
     ];
   ];

   urlParams = <|"status" → "success", "task" → "prep",
     "taskId" → Lookup[tasks, "taskId", ""], "protocolId" → protocol[ID],
     "samples" → (delSampleNamePrefix[#, prefix] & /@ samples),
     "sampleIds" → (Object[Container, Vessel, #][ID] & /@ samples),
     "time" → samplePrepEndTime|>;
   If[notifyServer[urlParams, extraOutputTag, maxRuns, waitInterval] =!= True,
    thisCycleWentFine = False;
    failMessage = <|"status" → "exception",
      "message" → "rememberSamplesNotifyServerException"|>;
   ];
   ,
   thisCycleWentFine = False;
   failMessage = <|"status" → "exception",
     "message" → "rememberSamplesException"|>;
  ];
 ];
```

3.2 - imageSamples (?)

```
If[thisCycleWentFine && StringMatchQ[taskType, "image", IgnoreCase → True],
 thisTaskGotMatched = True;
 urlParams = <|"status" → "exception", "message" → "unknownException"|>;

 {status, protocol, samples} = imageSamples[tasks, extraOutputTag];
 If[status =!= True,
  thisCycleWentFine = False;
  failMessage = <|"status" → "exception", "message" → "imageSamplesException"|>;
 ]
]
```

Supposedly, if the previous cell returns a valid protocol, it should automatically run. And the next cell will only start executing when that protocol has a status of completed.
NOTE: If the above cell (imageSamples) runs successfully, but the following cell (uploadSampleImage-ToServer) failed. You should go to a notebook page and fix uploadSampleImageToServer from there. DO NOT try to fix it in the script environment.

```
If[thisCycleWentFine && status && StringMatchQ[taskType,
    "image", IgnoreCase → True] && protocol[Status] === Completed,
  status =
   uploadSampleImageToServer[#, Lookup[tasks, "taskId", ""], extraOutputTag,
      validateImageTime, prefix, maxRuns, waitInterval] & /@ samples;
  If[AllTrue[status, TrueQ],
   urlParams = <|"status" → "success", "task" → "image",
     "taskId" → Lookup[tasks, "taskId", ""], "protocolId" → protocol[ID],
     "samples" → (delSampleNamePrefix[#, prefix] & /@ samples),
     "sampleIds" → (Object[Container, Vessel, #][ID] & /@ samples)|>;
   If[notifyServer[urlParams, extraOutputTag, maxRuns, waitInterval] =!= True,
    thisCycleWentFine = False;
    failMessage = <|"status" → "exception",
      "message" → "uploadSampleImageToServerNotifyServerException"|>;
   ];
   ,
   thisCycleWentFine = False;
   failMessage = <|"status" → "exception",
     "message" → "uploadSampleImageToServerException"|>;
  ];
 ];
```

3.3 - EOE (?)

```
If[thisCycleWentFine && StringMatchQ[taskType, "EOE", IgnoreCase → True],
 EOE = True;
 thisTaskGotMatched = True;
 Return[smartCellPrint[{"EOE string received at cycle ", whichCycle, "."},
   {"perm", "EOE", extraOutputTag}]]
]
```

3.4 - pause (?)

```
(* initialize some temporary variables *)
If[thisCycleWentFine && StringMatchQ[taskType, "pause", IgnoreCase → True],
  thisTaskGotMatched = True;
  whichCycle -= 1;
  donotuseme = 1;
  For[i = 1, i ≤ 3 * 60, i++,
   (* donotuseme=Mod[donotuseme*i+i,2024]; *)
   donotuseme =
     parseJsonResponse[URLRead[URLBuild[{globalDomainName, "pause"}]]];
  ];
  If[MemberQ[globalLogMgr[[-1, 1]], "line"],
   delCellWithStyle[{"perm", "line"}, Nothing, -1];
  ];
  If[! MemberQ[globalLogMgr[[-1, 1]], "pause"],
   printLineBreak["SCRIPT PAUSED", {"pause", extraOutputTag}];
  ];
 ];
```

3.5 - All other cases: thisTaskDidNotGotMatched?

what is going on on the server side???

```
If[thisCycleWentFine && thisTaskGotMatched =!= True,
  thisCycleWentFine = False;
  failMessage = <|"status" → "exception", "message" → "unknownTaskException"|>;
 ];
```

Exceptions? Errors? End of experiment?

```
If[thisCycleWentFine =!= True,
 pauseExecution[failMessage, extraOutputTag, maxRuns, waitInterval];
 smartCellPrint[{"Cycle ", whichCycle,
   " exited with an exception/error. Good luck fixing it!"},
  {"perm", extraOutputTag}];
 Return[printLogWithStyle[extraOutputTag]];,
 If[! StringMatchQ[taskType, "pause", IgnoreCase → True],
  smartCellPrint[{"Cycle ", whichCycle, " exited with success. Congrats!"},
   {"perm", extraOutputTag}];
  Return[printLogWithStyle[extraOutputTag]];
 ]
]
```

If you need to perform some task before going back to next loop, edit the following cell before it is too late!

```
(* your custom correction code starts here *)
useLastTask = False;
(* if you'd like to keep the last task, change to `True` *)

(* your custom correction code ends here *)

(* builtin error message cleaning code starts here*)
If[thisCycleWentFine =!= True,
  delCellWithStyle[{"threadExistsException", extraOutputTag}];
  delCellWithStyle[{"pauseExecution", extraOutputTag}];
  delCellWithStyle[{"getNextTaskFromServer", extraOutputTag}];
  delCellWithStyle[{"uploadSampleImageToServer", extraOutputTag}];
  delCellWithStyle[{"notifyServer", extraOutputTag}];
  delCellWithStyle[{"prepSamples", extraOutputTag}];
  delCellWithStyle[{"rememberSamples", extraOutputTag}];
  delCellWithStyle[{"imageSamples", extraOutputTag}];
 ];

(* delete the thread claimed task *)
If[Not[useLastTask],
  globalTaskMgr = delTaskForThread[thread];
 ];
```

Not end of experiment (EOE)? Go back to the start!

```
If[EOE =!= True,
  Goto["start:HSP"];,
  PauseScript[]; (* pause for final human confirmation *)
 ];
```

EOE at last! Let's print out the whole log.

```
extraOutputTag = "HSPMain:" <> thread;
printLineBreak["END", extraOutputTag];
smartCellPrint[{"End HSPMain (script) execution on thread `", thread,
   "`.\nPlease restart execution if needed be.\nThank you for using our
     automated HSP measurment script!"}, {"perm", extraOutputTag}];
```

```
log = printLogWithStyle[]
```