1. Create a new environment

Open your command prompt or terminal and navigate to the directory where you want to create the new environment.

PS: You can run terminal commands using !pip in Python if you are using a Jupyter Notebook or a similar environment. A command with ! in a Jupyter Notebook cell allows you to execute that command as if you were running it in a terminal.

```
python -m venv delta-statistic-python
```

Creating a new environment provides a controlled and isolated space for your projects, since each acts as a separate, self-contained workspace. This isolation helps avoid conflicts between different projects that may require different versions of the same package, allowing you to manage the dependencies of your project more effectively.

To activate the environment, use the following command:

- For Windows:

```
delta-statistic-python\Scripts\activate
```

- For MacOS or Linux:

```
source delta-statistic-python/bin/activate
```

2. Install and import Packages that are needed in this new

You can now install packages and libraries specific to this environment using "pip". If you only want to run a specific part of the code, you can install it one-by-one, otherwise it's simpler to run in the terminal:

```
pip install -r requirements.txt
```

Now simply import the needed packages to run the code:

```
import os, math, re
import numpy as np
import pandas as pd
from scipy.stats import entropy
from numba import njit, float64, int64
```

3. Import the delta-statistic (python) code

The delta function calculates the delta-statistic after an MCMC step. It uses the emcmc function to obtain Markov chain samples and then calculates the delta statistic. The [@njit] decorator is used for just-in-time compilation, which can improve the performance of the code.

To run it, first import the delta-statistic code available in Python. This can be done either by:

References

**Borges, R. et al. (2019). Measuring phylogenetic signal between categorical traits and phylogenies. Bioinformatics, 35, 1862-1869.<br>
[Article link](https://doi.org/10.1093/bioinformatics/bty800)
<br><br>
Diogo, R. (Github). Assessing traits and phylogenetic signal to unravel the tempo and mode of phenotypic evolution.<br>
[link](https://github.com/diogo-s-ribeiro/delta-statistic)**

- 3A. Importing from file

  Importing the provided delta-statistic file, if it is in the same working directory;

```
from delta_functs import delta
```

- 3B. Directly (Code)

  Copying all of the delta-statistic related functions.

```
# Metropolis-Hastings step for alpha parameter
@njit(float64(float64, float64, float64[::1], float64, float64))
def mhalpha(a,b,x,l0,se):
    '''a = The current value of the alpha parameter.
    b    = The current value of the beta parameter.
    x    = An array of data points used in the acceptance ratio computations, after uncertainty is calculated.
    l0   = A constant value used in the acceptance ratio computations.
    se   = The standard deviation used for the random walk in the Metropolis-Hastings algorithm.'''

    a1   = np.exp(np.random.normal(np.log(a),se, 1))[0]
    lp_a = np.exp( (len(x)*(math.lgamma(a1+b)-math.lgamma(a1)) - a1*(l0-np.sum(np.log(x)))) -
(len(x)*(math.lgamma(a+b)-math.lgamma(a)) - a*(l0-np.sum(np.log(x)))) )
    r    = min( 1, lp_a )

    # Repeat until a valid value is obtained
    while (np.isnan(lp_a) == True):
        a1   = np.exp(np.random.normal(np.log(a),se, 1))[0]
        lp_a = np.exp( (len(x)*(math.lgamma(a1+b)-math.lgamma(a1)) - a1*(l0-np.sum(np.log(x)))) -
(len(x)*(math.lgamma(a+b)-math.lgamma(a)) - a*(l0-np.sum(np.log(x)))) )
        r    = min( 1, lp_a )

    # Accept or reject based on the acceptance ratio
    if np.random.uniform(0,1) < r:
        return a1
    else:
        return a


# Metropolis-Hastings step for beta parameter
@njit(float64(float64, float64, float64[::1], float64, float64))
def mhbeta(a,b,x,l0,se):
    '''a = The current value of the alpha parameter.
    b    = The current value of the beta parameter.
    x    = An array of data points used in the acceptance ratio computations, after uncertainty is calculated.
    l0   = A constant value used in the acceptance ratio computations.
    se   = The standard deviation used for the random walk in the Metropolis-Hastings algorithm.'''

    b1   = np.exp(np.random.normal(np.log(b),se,1))[0]
    lp_b = np.exp( (len(x)*(math.lgamma(a+b1)-math.lgamma(b1)) - b1*(l0-np.sum(np.log(1-x)))) -
(len(x)*(math.lgamma(a+b)-math.lgamma(b)) - b*(l0-np.sum(np.log(1-x)))) )
    r    = min( 1, lp_b )

    # Repeat until a valid value is obtained
    while (np.isnan(lp_b) == True):
        b1   = np.exp(np.random.normal(np.log(b),se,1))[0]
        lp_b = np.exp( (len(x)*(math.lgamma(a+b1)-math.lgamma(b1)) - b1*(l0-np.sum(np.log(1-x)))) -
(len(x)*(math.lgamma(a+b)-math.lgamma(b)) - b*(l0-np.sum(np.log(1-x)))) )
        r    = min( 1, lp_b )

    # Accept or reject based on the acceptance ratio
    if np.random.uniform(0,1) < r:
        return b1
    else:
        return b


# Metropolis-Hastings algorithm using alpha and beta
@njit(float64[:, ::1](float64, float64, float64[::1], float64, float64, int64, int64, int64))
def emcmc(alpha,beta,x,l0,se,sim,thin,burn):
    '''alpha = The initial value of the alpha parameter.
    beta     = The initial value of the beta parameter.
    x        = An array of data points used in the acceptance ratio computations, after uncertainty is calculated.
    l0       = A constant value used in the acceptance ratio computations.
    se       = The standard deviation used for the random walk in the Metropolis-Hastings algorithm.
    sim      = The number of total iterations in the Markov Chain Monte Carlo (MCMC) simulation.
    thin     = The thinning parameter, i.e., the number of iterations to discard between saved samples.
    burn     = The number of burn-in iterations to discard at the beginning of the simulation.'''

    n_size = np.linspace(burn, sim, int((sim - burn) / thin + 1))
    usim   = np.round(n_size, 0, np.empty_like(n_size))
    gibbs  = []
    p      = 0

    for i in range(sim+1):
        alpha = mhalpha(alpha,beta,x,l0,se)
        beta  = mhbeta(alpha,beta,x,l0,se)

        if i == usim[p]:
            gibbs.append((alpha, beta))
            p += 1

    gibbs = np.asarray(gibbs)
    return gibbs


# Calculate uncertainty using different types
def entropy_type(prob, ent_type):
    '''prob  = A matrix of ancestral probabilities.
    ent_type = A string indicating the type of entropy calculation. (options: 'LSE', 'SE', or any other value for
Gini impurity).'''

    # Linear Shannon Entropy
    if ent_type == 'LSE':
        k    = np.shape(prob)[1]
        prob = np.asarray(np.where(prob<=(1/k), prob, prob/(1-k) - 1/(1-k)))
        tent = np.sum(prob, 1)

        # Ensure absolutes
        tent = np.asarray(np.where(tent != 0, tent, tent + np.random.uniform(0,1,1)/10000))
        tent = np.asarray(np.where(tent != 1, tent, tent - np.random.uniform(0,1,1)/10000))

        return tent

    # Shannon Entropy
    elif ent_type == 'SE':
        k    = np.shape(prob)[1]
        tent = entropy(prob, base=k, axis=1)

        # Ensure absolutes
        tent = np.asarray(np.where(tent != 0, tent, tent + np.random.uniform(0,1,1)/10000))
        tent = np.asarray(np.where(tent != 1, tent, tent - np.random.uniform(0,1,1)/10000))

        return tent

    # Ginni Impurity
    else:
        k    = np.shape(prob)[1]
        tent = ((1 - np.sum(prob**2, axis=1))*k)/ (k - 1)

        # Ensure absolutes
        tent = np.asarray(np.where(tent != 0, tent, tent + np.random.uniform(0,1,1)/10000))
        tent = np.asarray(np.where(tent != 1, tent, tent - np.random.uniform(0,1,1)/10000))

        return tent


# Calculate delta-statistic after an MCMC step
def delta(x,lambda0,se,sim,thin,burn,ent_type):
    '''x      = A matrix of ancestral probabilities.
    lambda0  = A constant value used in the acceptance ratio computations.
    se       = The standard deviation used for the random walk in the Metropolis-Hastings algorithm.
    sim      = The number of total iterations in the Markov Chain Monte Carlo (MCMC) simulation.
    thin     = The thinning parameter, i.e., the number of iterations to discard between saved samples.
    burn     = The number of burn-in iterations to discard at the beginning of the simulation.
    ent_type = A string specifying the type of entropy calculation (options: 'LSE', 'SE', or any other value for Gini
impurity).'''

    mc1    = emcmc(np.random.exponential(),np.random.exponential(),entropy_type(x,
ent_type),lambda0,se,sim,thin,burn)
    mc2    = emcmc(np.random.exponential(),np.random.exponential(),entropy_type(x,
ent_type),lambda0,se,sim,thin,burn)
    mchain = np.concatenate((mc1,mc2), axis=0)

    deltaA = (np.mean(mchain[:,1]))/(np.mean(mchain[:,0]))

    return deltaA
```

4. Ancestral Probabilities

The ancestral probabilities are needed before calculating their respective delta-statistic. This can be done either through Maximum Likelihood or Bayesian inference. Multiple software packages are currently available that can perform ancestral state reconstruction. Choose the best suited for your needs!

Since only the matrix with ancestral probabilities is needed, you can:

- 4A. Input them directly from a file, after calculating them through an external source

```
def read_file_ace(file_path, separator=',', rmv_col_name=True, rmv_row_name=True):
    ''' file_path = Represents the path to the file that you want to read.
    separator     (default: ',')  = Specifies the separator used in the file to separate the values.
    rmv_col_name  (default: True) = Boolean value that determines whether the first row (column names) should be
removed from the array.
    rmv_row_name  (default: True) = Boolean value that determines whether the first column (index) should be removed
from the array.'''

    # Read the data from the file using numpy's genfromtxt function
    array = np.genfromtxt(file_path, delimiter = separator)

    # Check if the column name should be removed
    if rmv_col_name == True:
        array = np.delete(array, 0, axis=0)     # Remove the first row (Trait Name)

    # Check if the row name should be removed
    if rmv_row_name == True:
        array = np.delete(array, 0, axis=1)     # Remove the first column (Entity Name)

    return array
```

- 4B. Or use a package to calculate them directly in Python (e.g.)

    - 4B.1. PastML

A good package for Ancestral Character Estimation (through Maximum Likelihood) in Python is PastML. If you intend to run PastML don't forget to install the package.

    - 4B.2. rpy2:ape

Another popular option to calculate the marginal probabilities would be to use the ape package in Python through rpy2.
Once again, don't forget to install the necessary packages.

PS. If the ancestral probabilities matrix can't properly be calculated, resulting in a matrix with multiple **-NaN-** values, there might be a problem related to the tree branch lengths. Try uncommenting the "tree$edge.length" line and rerun the code.

5. Calculate Delta-Statistic

```
lambda0  = 0.1                          # rate parameter of the proposal
se       = 0.5                          # standard deviation of the proposal
sim      = 100000                       # number of iterations
thin     = 10                           # Keep only each xth iterate
burn     = 100                          # Burned-in iterates

ent_type = 'LSE'                        # Linear Shannon Entropy
```

It is also possible to calculate uncertainty using a normalized version of both:

1) The Shannon Entropy: &emsp; A widely used measure of information content or uncertainty in a random variable;
2) The Gini impurity: &emsp;&emsp;&emsp; Measure that can be used to quantify the impurity or disorder in a set of class labels.

- **ACE (see above "4. Ancestral Probabilities") and Delta Calculation**

Before starting, make sure that:

- The necessary files are in the correct directory;
- The functions of the preferred ACE and delta calculation have already run.

A1) (Single) Input directly

```
path_ap      = r"./input/Simplified/Ancestral_Probabilities/FILE"      # Path to the (ap: ancestral probabilities)
input files

file         = "Simplified_AP_1.txt"                                   # File with Ancestral Probabilities
file         = path_ap.replace('FILE', file)                           # Path + file

Ancest_Prob  = read_file_ace(file_path=file, separator=',', rmv_col_name=True, rmv_row_name=True)
Delta_Final  = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin, ent_type='LSE')

print( Delta_Final )
```

A2) (Multiple) Input directly

```
def multiple_files_ap( dir=None, separator=',', rmv_col_name=True, rmv_row_name=True, lambda0=0.1, se=0.5, 
sim=100000, thin=10, burn=100, ent_type='LSE' ):
    '''dir          (default: None)   = Represents the directory path with the ancestral probabilities matrices. If no 
value is provided when calling the function, the user will be prompted to input the directory path.
    separator    (default: ',')    = Determines the separator used in the matrices when reading the files.
    rmv_col_name (default: True)   = Boolean value that indicates whether the column names should be removed when 
reading the files.
    rmv_row_name (default: True)   = Boolean value that indicates whether the row names should be removed when 
reading the files.
    lambda0      (default: 0.1)    = A constant value used in the acceptance ratio computations.
    se           (default: 0.5)    = The standard deviation used for the random walk in the Metropolis-Hastings 
algorithm.
    sim          (default: 100000) = The number of total iterations in the Markov Chain Monte Carlo (MCMC) 
simulation.
    thin         (default: 10)     = The thinning parameter, i.e., the number of iterations to discard between saved 
samples.
    burn         (default: 100)    = The number of burn-in iterations to discard at the beginning of the simulation.
    ent_type     (default: 'LSE')  = A string specifying the type of entropy calculation (options: 'LSE', 'SE', or 
any other value for Gini impurity).'''

    if dir == None:
        dir = str(input( 'What is the directory path with the ancestral probabilities matrices?' ))

    dic_files = set( os.listdir(dir) )
    dic_delta = set()
    for file in dic_files:
        file = dir + file
        ace = read_file_ace( file, separator, rmv_col_name, rmv_row_name )
        dic_delta.add( delta(x=ace, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin, ent_type=ent_type) )

    return dict(zip( dic_files, dic_delta ))
```

```
Delta_Final_dic = multiple_files_ap( dir = './input/Simplified/Ancestral_Probabilities/' )
```

```
print( Delta_Final_dic )
print( Delta_Final_dic.values() )
```

B1) (Single) PastML

```
path_data  = r"./input/3Class/3C_States.txt"                # File containing tip/node annotations, in csv or tab
format
path_tree  = r"./input/3Class/Trees/3C_Trees_1.txt"         # File containing tip/node annotations, in csv or tab
format

method     = "MPPA"                                         # MPPA, MAP
model      = "F81"                                          # F81, JC, EFT

Ancest_Prob = marginal(path_tree, path_data)
Delta_Final = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin, ent_type='LSE')

print(Delta_Final)
```

B2) (Multiple) PastML

```
def multiple_files_PastML( dir_tree=None, dir_data=None, single_tree_file=False, threads=0, lambda0=0.1, se=0.5,
sim=100000, thin=10, burn=100, ent_type='LSE' ):
    '''dir_tree      (default: None)   = Represents the directory path to the phylogenetic tree(s) file(s). If no
value is provided when calling the function, the user will be prompted to input the directory path.
    dir_data         (default: None)   = Represents the directory path to the observed states file(s). If no value is
provided when calling the function, the user will be prompted to input the directory path.
    single_tree_file (default: False)  = Boolean value that indicates whether the phylogenetic tree(s) file(s) are
stored in a single file (True) or multiple files (False).
    threads          (default: None)   = This variable represents the number of threads used in the marginal
function.
    lambda0          (default: 0.1)    = A constant value used in the acceptance ratio computations.
    se               (default: 0.5)    = The standard deviation used for the random walk in the Metropolis-Hastings
algorithm.
```

```
    sim              (default: 100000) = The number of total iterations in the Markov Chain Monte Carlo (MCMC)
simulation.
    thin             (default: 10)     = The thinning parameter, i.e., the number of iterations to discard between
saved samples.
    burn             (default: 100)    = The number of burn-in iterations to discard at the beginning of the
simulation.
    ent_type         (default: 'LSE')  = A string specifying the type of entropy calculation (options: 'LSE', 'SE',
or any other value for Gini impurity).'''

    if dir_tree == None:
        dir_tree = str(input( 'What is the directory path to the phylogenetic tree(s) file(s)?' ))
    if dir_data == None:
        dir_data = str(input( 'What is the directory path to the observed states file(s)?' ))

    if single_tree_file==False:
        dic_files = set( os.listdir( dir_tree ) )
        dic_delta = set()
        for phylo_tree in dic_files:
            Ancest_Prob  = marginal( dir_tree+'/'+phylo_tree, dir_data, single_tree_file=single_tree_file,
threads=threads )
            dic_delta.add( delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin,
ent_type=ent_type) )
        dic_final = dict(zip( dic_files, dic_delta ))
    else:
        dic_final = {}
        arr       = np.loadtxt( dir_tree , dtype=str)
        indx      = 0
        for phylo_tree in arr:
            Ancest_Prob = marginal( phylo_tree, dir_data, single_tree_file=single_tree_file, threads=threads )
            delta_value = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin,
ent_type=ent_type)

            phylo_tree  = 'Tree:' + str(indx) + '_' + phylo_tree
            dic_final[phylo_tree] = delta_value

            indx += 1
    return dic_final
```

```
path_tree  = r"./input/3Class/Trees"
Delta_Final = multiple_files_PastML(path_tree, path_data, single_tree_file=False)

print( Delta_Final )

path_tree  = r"./input/3Class/3C_Multiple_Trees.txt"
Delta_Final = multiple_files_PastML(path_tree, path_data, single_tree_file=True)

print( Delta_Final )
print( Delta_Final.values() )
```

C1) (Single) rpy2:ape

PS. Before starting don't forget to install ape with the variable "install_ape = True"

```
path_tree = "./input/2Class/Trees/2C_Trees_1.txt"
path_data = "./input/2Class/2C_States.txt"

Ancest_Prob = np.asarray( to_ape(path_tree, path_data, instal_ape=True, tree_dir=True) )
Delta_Final = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin, ent_type='LSE')

print( Delta_Final )
```

C2) (Multiple) rpy2:ape

PS. Before starting don't forget to install ape with the variable "install_ape = True"

```
def multiple_files_Ape( dir_tree=None, dir_data=None, single_tree_file=False, lambda0=0.1, se=0.5, sim=100000,
thin=10, burn=100, ent_type='LSE', install_ape=False ):
    '''dir_tree      (default: None)   = Represents the directory path to the phylogenetic tree(s) file(s). If no
value is provided when calling the function, the user will be prompted to input the directory path.
```

```
    dir_data         (default: None)   = Represents the directory path to the observed states file(s). If no value is
provided when calling the function, the user will be prompted to input the directory path.
    single_tree_file (default: False)  = Boolean value that indicates whether the phylogenetic tree(s) file(s) are
stored in a single file (True) or multiple files (False).
    lambda0          (default: 0.1)    = A constant value used in the acceptance ratio computations.
    se               (default: 0.5)    = The standard deviation used for the random walk in the Metropolis-Hastings
algorithm.
    sim              (default: 100000) = The number of total iterations in the Markov Chain Monte Carlo (MCMC)
simulation.
    thin             (default: 10)     = The thinning parameter, i.e., the number of iterations to discard between
saved samples.
    burn             (default: 100)    = The number of burn-in iterations to discard at the beginning of the
simulation.
    ent_type         (default: 'LSE')  = A string specifying the type of entropy calculation (options: 'LSE', 'SE',
or any other value for Gini impurity).
    instal_ape       (default: False)  = Boolean value that determines whether to install the 'ape' package or
not.'''

    if dir_tree == None:
        dir_tree = str(input( 'What is the directory path to the phylogenetic tree(s) file(s)?' ))
    if dir_data == None:
        dir_data = str(input( 'What is the directory path to the observed states file(s)?' ))

    if install_ape==True:
        r(
        """
        # Instal Ape
        install.packages('ape')
        """)

    if single_tree_file==False:
        dic_files = set( os.listdir( dir_tree ) )
        dic_delta = set()
        for phylo_tree in dic_files:
            dir_file     = np.loadtxt( dir_tree + '/' + phylo_tree, dtype=str )
            Ancest_Prob  = np.asarray( to_ape( dir_file, dir_data, instal_ape=False ) )
            dic_delta.add( delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin,
ent_type=ent_type) )
```

```
        dic_final = dict(zip( dic_files, dic_delta ))

    else:
        dic_final = {}
        arr       = np.loadtxt( dir_tree , dtype=str)
        indx      = 0
        for phylo_tree in arr:
            Ancest_Prob = np.asarray( to_ape( phylo_tree, dir_data, instal_ape=False ) )
            delta_value = delta(x=Ancest_Prob, lambda0=lambda0, se=se, sim=sim, burn=burn, thin=thin,
ent_type=ent_type)

            phylo_tree  = 'Tree:' + str(indx) + '_' + phylo_tree
            dic_final[phylo_tree] = delta_value

            indx += 1

    return dic_final
```

```
path_tree = './input/2Class/2C_Multiple_Trees.txt'
Delta_Final = multiple_files_Ape( path_tree, path_data, single_tree_file=True)

print( Delta_Final )
print( Delta_Final.values() )

path_tree = './input/2Class/Trees/'

Delta_Final = multiple_files_Ape( path_tree, path_data, single_tree_file=False)

print( Delta_Final )
print( Delta_Final.values() )
```