Implementing Dirichlet boundary conditions and selecting boundary vertices in Python

Experience teaches us that the students tend to struggle with the implementation of the (if present) Dirichlet boundary conditions in their Python code.
While we provided you with a function that solves a linear problem subject to Dirichlet data in this year's iteration of the course (i.e. `solve_with_dirichlet_data` in `solve.py`) , several people have approached me and asked me to provide a detailed explanation of the concept, which I will do in this document.

Implementing BCs on a discrete level

While there is no unique way to strongly enfore Dirichlet data, there exists one way that should always work. Implementing Dirichlet BCs can be cumbersome without this trick but becomes manageable once you know it.
As a downside, this approach may not always be the most efficient choice but since we do not focus on efficient implementations in this course, **we encourage you to use this approach as much as possible**, as it will minimise the difficult-to-debug coding mistakes associated with the BCs.

Assume we are confronted with a PDE whose weak form is associated with a bilinear form $a(\cdot, \cdot)$ and a right hand side $f(\cdot)$. There may be a nonempty Neumann-boundary Γ_N whose associated Neumann data is absorbed into $a(\cdot, \cdot)$ and $f(\cdot)$. Assume that the Dirichlet boundary $\Gamma_D \neq \emptyset$.
Suppose you have implemented $a_h(\cdot, \cdot)$ and $f_h(\cdot)$, the discretised counterparts of $a(\cdot, \cdot)$ and $f(\cdot)$ in your code, i.e., your code is capable of evaluating $a_h(\phi_i, \phi_j)$ and $f_h(\phi_i)$ for any choice of $(\phi_i, \phi_j) \in \mathcal{V}_h \times \mathcal{V}_h$ and $\phi_i \in \mathcal{V}_h$, where $\mathcal{V}_h$ denotes your finite-dimensional basis (in this course, typically $\mathbb{P}_1$ over your mesh).
By A, we denote the matrix with entries $A_{ij} = a_h(\phi_i, \phi_j)$ **over all** $(\phi_i, \phi_j) \in \mathcal{V}_h \times \mathcal{V}_h$, **i.e., including those that are nonvanishing on** Γ_D. Similarly, $\mathbf{f}$ denotes the discrete load vector under the same conditions. Your code should be capable of assembling both A and $\mathbf{f}$ for you. If there were no Dirichlet bundary, obtaining the solution vector $\mathbf{u}$ would amount to solving

$$A\mathbf{u} = \mathbf{f}. \tag{1}$$

Now assume that on the Dirichlet boundary Γ_D, we have $u(\mathbf{x}) = u_D$. Given A, $\mathbf{f}$, how do we enforce that in our solution vector ?
The first ingredient we need is the index-set of boundary nodes, i.e., the collection of indices $\mathcal{I}_{\text{boundary}} = \{i \mid \phi_i|_{\Gamma_D} \neq 0\}$. You will find a concrete example of how to obtain this index-set in the `examples.py` of the Python code. Given u_D, we now require $\mathbf{u}_i = u_D(\mathbf{x}_i), \ \forall i \in \mathcal{I}_{\text{boundary}}$, where $\mathbf{x}_i \in \Gamma_D$ is the vertex-position associated with $\phi_i \in \mathcal{V}_h$. Without loss of generality, let us

assume that the $i \in \mathcal{I}_{\text{boundary}}$ are ordered such that we can decompose

$$\mathbf{u} = \begin{bmatrix} \mathbf{u}_0 \\ \mathbf{u}_D \end{bmatrix}, \tag{2}$$

where $\mathbf{u}_0$ is the (unknown) vector of inner weights, while $\mathbf{u}_D$ is a vector containing the discrete evaluations of the Dirichlet data $u_D(\mathbf{x}_i)$ in the boundary vertices $\mathbf{x}_i \in \Gamma_D$. Similarly, we can decompose A and $\mathbf{f}$

$$A = \begin{bmatrix} \tilde{A} & B \\ C & D \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} \mathbf{f}_0 \\ \mathbf{f}_D \end{bmatrix} \tag{3}$$

into appropriately-sized blocks whose dimensions match the decomposition of $\mathbf{u}$. In the presence of a Dirichlet boundary, we only test against ϕ_i with $i \notin \mathcal{I}_{\text{boundary}}$. As A and $\mathbf{f}$ result from evaluating $a_h(\phi_i, \phi_j)$ and $f_h(\phi_i)$ for all ϕ_i, the reduced system matrix and load vector simply results from slicing out the bottom block-row of A and $\mathbf{f}$

$$\begin{aligned} \begin{bmatrix} \tilde{A} & B \\ C & D \end{bmatrix} \begin{bmatrix} \mathbf{u}_0 \\ \mathbf{u}_D \end{bmatrix} = \begin{bmatrix} \mathbf{f}_0 \\ \mathbf{f}_D \end{bmatrix} \quad \text{becomes} \quad \begin{bmatrix} \tilde{A} & B \end{bmatrix} \begin{bmatrix} \mathbf{u}_0 \\ \mathbf{u}_D \end{bmatrix} = \mathbf{f}_0 \\ \implies \tilde{A}\mathbf{u}_0 = \mathbf{f}_0 - B\mathbf{u}_D = \tilde{\mathbf{f}}. \end{aligned} \tag{4}$$

Hence, we implement the BCs by replacing $A \to \tilde{A}$, $\mathbf{f} \to \tilde{\mathbf{f}} = \mathbf{f}_0 - B\mathbf{u}_D$ and solving for the sub-vector $\mathbf{u}_0$ of $\mathbf{u}$.
In your Python code, you can generally not assume that you can decompose $\mathbf{u}$ in the way sketched above. However, given an (ordered) array containing the indices $i \in \mathcal{I}_{\text{boundary}}$, you can accomplish the same by using scipy / numpy's slicing functionality. The function `solve_with_dirichlet_data` has been updated with many additional comments that explain the various sub-steps of this document's explanation. If you require more assistance, feel free to ask questions online or come to Friday's exercise class. The staff will be more than happy to help.